Behavioural Semantics for Asynchronous Components

R. Ameur-Boulifa[a], L. Henrio[b,*], O. Kulankhina[c], E. Madelaine[c,*], A. Savu[c]

[a] *LTCI, Télécom ParisTech, Univ. Paris-Saclay, 75013, Paris, France*
[b] *Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, France.*
[c] *INRIA-I3S-CNRS, University of Nice Sophia Antipolis, France*

Abstract

Software components are a valuable programming abstraction that enables a compositional design of complex applications. In distributed systems, components can also be used to provide an abstraction of locations: each component is a unit of deployment that can be placed on a different machine. In this article, we consider this kind of distributed components that are additionally loosely coupled and communicate by asynchronous invocations.

Components also provide a convenient abstraction for verifying the correct behaviour of systems: they provide structuring entities easing the correctness verification. This article provides a formal background for the generation of behavioural semantics for asynchronous components. It expresses the semantics of hierarchical distributed components communicating asynchronously by requests, futures, and replies; this semantics is provided using the pNet intermediate language. This article both demonstrates the expressiveness of the pNet model and formally specifies the complete process of the generation of a behavioural model for a distributed component system. The purpose of our behavioural semantics is to allow for verification both by finite instantiation and model-checking, and by techniques for infinite systems.

Keywords: Behavioural specification, software components, distributed systems, futures

1. Introduction

Ensuring the safety of distributed applications is a challenging task. Both the network and the underlying infrastructure are not reliable, and additionally, even without failures, applications are complicated to design because of

*Corresponding author
Email addresses: rabea.ameur-boulifa@telecom-paristech.fr (R. Ameur-Boulifa), ludovic.henrio@cnrs.fr (L. Henrio), oleksandra.kulankhina@inria.fr (O. Kulankhina), eric.madelaine@inria.fr (E. Madelaine)

the multiple execution paths possible. To ensure the safety of distributed applications, we propose to use formal methods to be able to verify the correct behaviour of distributed applications. Consequently, it is necessary to choose a programming abstraction that is convenient to write applications, but also that provides enough information to be able to check the properties of the program. We adopt a programming model that is expressive enough to program complex distributed applications but with some constraints enabling the behavioural verification of these applications. This programming model relies on the notion of distributed software components. Component models provide a structured programming paradigm, and ensure a very good re-usability of programs. Indeed in component applications, both required and provided functionalities are defined by means of provided/required ports; this improves the program specification and thus its re-usability. Several effective distributed component models have been specified, developed, and implemented in the last years [1, 2, 3, 4] ensuring different kinds of properties to their users. Component models have been chosen as the target programming model for many developments in formal methods. Indeed, additionally to the valuable software engineering methodology they ensure, components also provide structural information that facilitate the use of formal methods: the architecture of the application is defined statically.

Distributed component models. Even if our theorems and results can be adapted to most component models, we primarily focus on one distributed component model, the GCM (Grid Component Model [4]). This component model originates from the Grid computing community, it is particularly targeted at composing large-scale distributed applications. Its reference implementation GCM/ProActive relies on the notion of active objects, and ensures that, during execution, each thread is isolated in a single component. Because of active objects, components that usually structure the application composition also provide the structure of the application at runtime, in terms of location and thread (a single applicative thread manipulates the state of the component). We call this kind of components *asynchronous components* because they are loosely coupled entities communicating by a request-reply mechanism. All those aspects facilitate the use of formal methods for ensuring safe behaviour of applications, but they also require specific developments to produce a formal model of an application built from such components.

Contribution. This article formalises the construction of a behavioural model for the core constructs of GCM/ProActive components. It describes formally how we can generate a behavioural model in terms of *pNets* (parameterised Networks of synchronised automata) [5] from the description of the architecture of a GCM/ProActive application and the description of the behaviour of each service method implemented by the programmer. pNets are networks of LTSs with parameters organised (and synchronised) in a hierarchical way. In this article we formalise the automatic construction of the behavioural model for communication, management, and composition aspects.

Our behavioural models are *parameterised*: pNets specify a structured composition of labelled transition systems (LTS) that can use parameters/variables. Each pNet is either formed of other pNets or is a single LTS. Parameters can be used as local variables in an LTS; but they can also be used to define families of pNets of variable size, and to specify synchronisation between pNets (see Section 4). Once the parameterised behavioural model is generated, we can for example generate a finite instance of the model that can be checked against correctness formulas using a model-checking tool. But our behavioural model is richer than what can be checked by finite-state model-checkers and other verification techniques can also be used. For example we are currently working on more symbolic techniques mixing bisimulation algorithms with satisfiability engines to deal with properties of pNet systems with unbounded data, or with unknown sub-nets [6]).

The GCM model and its GCM/ProActive implementation provide a very rich environment for building distributed applications, including too many features to be fully formalised in this article. In section 2, we shall define formally a "*Core GCM*" model, containing its most important constructs, namely:

- Primitive components: at the leaves of the hierarchy, from the definition of the service methods, we specify a component able to receive requests and serve each of them one after the other. When a request service terminates, a reply is sent back to the component that emitted the request. The crucial parts composing the model of a primitive component are: the request queue, the handling of asynchronous communications for sending requests and replies, the futures and their management (Section 4.1).

- Composite components (composites, for short): as our component model is hierarchical, a component can be built from the composition of other components. In GCM, composites are instantiated at runtime, it is thus necessary to specify their behaviour in our model too (see Section 4.2). Each composite is in fact implemented as an active object and thus the internal structure of a composite is very similar to the one of a primitive component.

- Component composition: from an ADL (architecture description language) specification, we generate the synchronisations corresponding to the communications that can occur between the different components.

- Futures: futures are frequently used in active object languages, they are place-holders for results of asynchronous invocations, called requests here. We encode in our models the mechanisms for dealing with futures (Section 4.1.5) and the transmission of futures references between components (*first class futures*, Section 5).

Previous works. This article is built upon previous works of the authors. The pNets have already been defined formally in [5], but in a more complex version using a specific form of controllers named "transducers". In this article, in Section 3, we shall provide a new definition, simpler and more concise.

The modelling of basic component features has been addressed in previous publications, including: 1) the pNets-based semantics of primitive and composite components, and their hierarchical composition have been described in [7] and [5]; 2) behavioural models for first-class futures have been studied in [8].

In these works, we built pNet models for specific features of GCM in the context of specific case-studies and proved properties of the studied applications. To illustrate the kind of properties we are able to verify, we proved by model-checking that a master-slave fault-tolerant application [9] behaves correctly: 1) it answers to requests: we proved both reachability and (fair) inevitability of termination of services, 2) the answers (values returned by services) are correct. This shows that our approach addresses both safety and liveness properties, but also functional correctness, modulo data abstraction, based on user requirements.

In Sections 4 and 5 we provide a general formalization of the features previously studied in those examples.

Full GCM. We have defined our *Core GCM* as containing the minimal set of features needed to build interesting hierarchical asynchronous distributed components, including the future mechanism ensuring transparent wait-by-necessity request calls. Previous works also addressed advanced features of the GCM, available in the GCM/ProActive middleware, that we will not address in this article. This constitutes what we call *Full GCM*, and includes:

- Stateful components in [10]
- Non-functional concerns, as a full-fledge componentisation of the component membranes. This is addressed in [10]
- Reconfiguration and reconfiguration management: some early version of binding controllers were defined in [5]; and an industrial-inspired case-study with full reconfiguration features was studied in [11]
- Modelling of group communication, collective interfaces, and collective interface policies; this was defined in [12]

This article focuses on the static structure of component systems but contains all the building blocks to reason on these extended features. We do not include the semantics of these extensions here, for reasons of space, and because it would add still more complexity to the formalisation. But the Core is easily extensible, and a number of the Full GCM features are implemented in the VerCors platform (see [10]).

Compared to those previous works, this article first proposes a model aggregating all those features of the Core GCM component model. More important, this article fully formalises the modelling process in a systematic way compared to the previous case-studies. This is a necessary step for the automatic generation of behavioural models for component systems. Also this article is the first one to propose a model for the handling of futures in composite components.

Overall this article builds upon the individual case studies considered in our previous works to fully formalise the generation of behavioural models for asynchronous components. The different results mentioned above should be considered also as a validation of our approach, they show that the semantics we define in this article allowed us to prove the correctness of various applications.

The objective of this article is not to define a reduction semantics of GCM, such a semantics has already been published and formalised in Isabelle/HOL [13, 14]. This semantics was particularly convenient to reason about generic properties of the component model and to prove generic properties valid for any GCM application. The purpose in this article is rather to define a semantics of GCM as an intermediate model suitable for (automatic) verification of application properties, like model-checking for example. In this article we focus on the exhaustive definition of the behavioural semantics for the whole GCM model. The operational semantics provided in [13, 14] was high-level and abstract enough to manage easily proofs on paper and in a theorem prover. It used high-level constructs (maps, lists, sets) to formalize futures and request queues for example. The behavioural semantics presented here is at the right level for automated reasoning in tools like model-checkers, it is much more operational. These differences make the proof of equivalence between the two semantics out of scope for the present paper. However we will briefly explain how such a proof could be performed in Section 6. We consider that our semantics is validated by 1) the different use-cases published in previous works; 2) a set of theorems showing that our semantics is somehow well-formed in Section 6.1; 3) a full example illustrating most of the semantic rules defined in this article in Section 6.3; 4) the tool support provided by the VerCors environment. In Section 7 we discuss briefly two main features of Full GCM, namely state-full components and reconfiguration.

Why pNets. pNets provide two characteristics that are crucial in this work. First pNets are hierarchically structured. This makes the structure of the formalisation similar to that of the components we model. Interactions can then be specified at the same level and between the same entities as they occur in GCM programs; for this reason there is no need for a restriction operator in pNets because each composition operator controls locally the interactions that occur. This also makes the approach compositional. Second, pNets are parameterised and parameters can not only be used as data and inside processes, but also synchronisation between processes can be dependent on parameters. Typically, synchronisation vectors allow one to express easily the fact that an emitter process sends a message $m(i)$ and that this message m is received by the i^{th} process of a family of receivers. To the best of our knowledge, only pNets allow such flexibility in the handling of parameters, indexes and processes without renaming actions nor using ad-hoc synchronisation primitives. This feature is crucial to encode easily families of methods, components, and futures. Interaction with elements of these families is directly based on the source/target index and does not rely on the creation of artificial communication channels.

Of course such expressiveness does not come for free. The counterpart we

have to pay is to define a rich and specific syntax and semantics for our pNets, which somehow explains the complexity of the notations we use in this article.

Beyond GCM. Our behavioural specification is particularly adapted to the reasoning on GCM components, however, the approach is applicable to a wider spectrum of programming models. The component structure of GCM is quite similar to the one of Fractal [15] and SCA [3]; the runtime behaviour of components uses active objects/actor-like computations, which is similar to Creol [16], AmbientTalk [17], JCobox [18], ABS [19], and SynchNet [20]. More precisely, the structure we create can be used to represent the SCA component structure but the complexity of the runtime support provided by the SCA framework would require us to formally specify many additional entities. The future proxies we model can be reused, as well as the request queues, to encode JCobox or ABS active objects. The explicit nature of futures in JCobox and ABS would make the handling of proxies and of futures easier, however the cooperative scheduling of requests would require a few additional synchronisation constructs in these languages. The simplicity of our model fits well with the coordination language SynchNet; the coordination mechanism could be specified by a combination of synchronisation constructs and controllers. Consequently, the approach and results presented in this article can be adapted to the behavioural specification of systems using these component and programming models and we believe that our contribution provides a promising approach for specifying such component systems.

Structure of the paper. This article is organised as follows. Section 2 gives a brief overview of the GCM component model defined in [4]; it also gives the abstract syntax we use for the definition of component systems and defines the set of components we consider as *well-formed* in Section 2.3. Section 3 provides a definition of the pNets formalism [5]. Then Sections 4 and 5 contain the main contribution of the article; they present respectively the basic behavioural model for GCM components and a more complex feature: first-class futures. Section 6 acts as a validation of our approach; it consists of two correctness theorems in Section 6.1, and an example showing how the model is built in Section 6.3. This article concludes with a comparison with related works in Section 8 and a conclusion. Appendices describe the semantics of pNets and a summary of main behavioural semantic functions.

2. The Grid Component Model: GCM

GCM has been proposed in the CoreGrid Network of Excellence, it is an extension of the Fractal component model [21, 22] to better address large-scale distributed computing. GCM builds above Fractal and thus inherits its hierarchical structure, the enforcement of separation between functional and non-functional concerns, its extensibility, and the separation between interfaces and implementation.

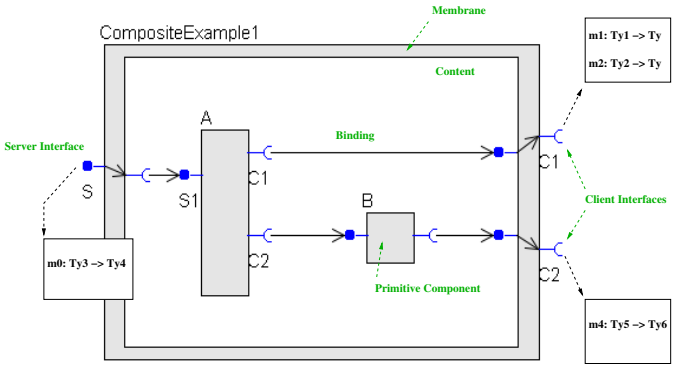

Figure 1: A typical GCM assembly

Figure 1 shows the basic component structure provided by GCM. It introduces the terminology used to describe GCM components and their composition. Composite components are made of one or several sub-components, while primitive ones are black boxes. Components expose interfaces that are bound together when assembling components into a composite. Client interfaces emit invocations; server interfaces receive invocations. Interfaces are annotated with their signature, i.e. the type of their methods. Each message emitted (resp. received) by an interface must be the invocation (resp. the reception of an invocation) on a method invocation belonging to the interface signature.

GCM has been conceived thinking of components with an intermediate size, i.e. the typical size of an MPI process, though it can be used to define much finer or coarser grain components. In GCM/ProActive the primitive components (and the composite ones too) have by nature this intermediate size: they contain an activity, i.e. an active object, its dependencies, a request queue, and a thread. This article relies on the fact that components are used as structuring entities that specify the different threads of the applications and the set of objects manipulated by each of those threads. It is very interesting and convenient to use formal methods to check the properties of GCM applications, as the component structure gives crucial information on the concurrency that occurs at runtime.

GCM and Fractal come with an ADL (architecture description language) providing a textual (XML-based) way of describing component assembly. Such a description of the application architecture (either textual or graphical) is the starting point of our work. Starting from the architecture description and the description of the behaviour of each service method, we build a formal description of the behaviour of the whole application.

2.1. A reference implementation for GCM

GCM/ProActive is a reference implementation of the GCM component model. It is based on the ProActive Java library and relies on the notion of active ob-

jects. It is important to note that each component corresponds at runtime to an active object and consequently each component can easily be deployed on a separate JVM and can be migrated.

One of the main advantages of using active objects to implement components is their adaptation to distribution. Indeed, by nature active objects provide a natural way to provide loosely coupled components. By loosely coupled components, we mean components responsible for their own state and evaluation, and only communicating via requests and replies. Asynchronous requests increase and automate parallelism, and absence of sharing eases the design of concurrent systems. Additionally, loose coupling reduces the impact of latency, and limits the interleaving between components. Finally, independent components also ease the autonomic management of component systems, enabling systems to be more dynamic, more scalable, and easily adaptable to different execution contexts. This is why we think that active objects are particularly adapted to implement a distributed component model.

2.2. Informal semantics of asynchronous components

This section describes briefly an informal semantics of GCM/ProActive components. The general principle is that interaction between components is limited to communications, and more precisely to a request/reply mechanism. Communicating by asynchronous requests allows each component to execute asynchronously from the others. However it is commonly necessary to obtain a result for some of those asynchronous invocations. A convenient abstraction for dealing with response to asynchronous requests is the notion of futures. A formal and general semantics of GCM/ProActive can be found in [13].

Communications. The basic communication paradigm we consider is asynchronous message sending: upon a communication the message is enqueued at the receiver side in a queue. To prevent memory from being shared between components, messages can only transmit parameters which are copied at the receiver side; no object or component can be passed by reference. This communication semantics is similar to message passing in the Actor model [23]. The messages sent between components are called *requests*. We call our component model asynchronous because communication does not trigger computation on the receiver side immediately, it just enqueues a request. In practice the communication itself, i.e. the en-queueing of the request is synchronised: a brief rendez-vous ensures that the sender only continues when the request is received at destination; according to [24] this ensures causal ordering of messages in the sense of [25]. Such a synchronisation phase is not optimal in terms of parallelism but greatly eases programming.

Synchronisation on request results relies on futures that are place-holders for objects yet-to-be-computed. Futures come with a natural synchronisation construct. Accessing a future enforces a synchronisation at a certain program point that can only be released when the future is given a value. Those synchronisation points may sometimes block a component. However, synchronisation on future avoids the inversion of control featured by languages using some form

of asynchronous callback mechanisms (e.g. tokens in SALSA [26], AmbientTalk futures [17], callback on futures in Akka). In other words asynchronous callbacks are more asynchronous but sometimes make the programming of applications less intuitive. We say that futures are first class if future references can be transmitted between remote entities without requiring the future to be resolved. First class transparent futures are unique in ASP; they are transparent to the user and ensure that the synchronisation on the future is triggered automatically and only when the method result is strictly needed. The synchronisation on futures guarantees some confluence properties on ASP [27]. A comparison between different distributed languages and the way they handle futures can be found in [28].

To allow for transparent asynchronous requests with results, we use transparent first-class futures. The promise for a reply to a request is created automatically when the request is sent, we call it a *future*. For accessing the value of a future, the caller must wait until the request is treated and the result sent. When the request is finished, the result is automatically sent to replace all the references to the corresponding future. Futures are said to be *first-class* if they can be transmitted between components.

Component behaviour. The primitive components encapsulate the business code. They generally serve requests in the order of arrival, providing answer for all the requests they receive. They can call other components by emitting a request on one of the client interfaces. In GCM/ProActive, each component is mono-threaded: a single request is served at a time and no internal concurrency occurs in a component. However, a component always accepts the reception of a future value, or of a request.

While primitive components contain the application logic, composite components have a predefined behaviour because they are only used as composition tools. Composites serve requests in a FIFO order, delegating requests to the bound component, which can be either sub-components or external ones. A composite performs no computation: it only delegates requests.

Components featuring the semantics defined above are loosely coupled; they are better adapted to a distributed setting and easier to program safely because of the limited concurrency they allow. However, the semantic of such components rely on several notions, like for example request queues and futures, that have to be specified when building a behavioural model for those components. This article specifies how those notions can be formally defined as pNets, and how those definitions can be used to build the behavioural model of a component-based application.

2.3. Component Definition

This section defines a hierarchical structure for representing components. We define a syntax for describing the different elements of a GCM component assembly. We also define a set of auxiliary functions that will help us to manipulate the component structure, and finally we define what component systems

we consider to be well-formed, these are the systems for which we are able to build a behavioural model. These definitions collect the information contained in the ADL and in the source code that will be useful for our analysis; they also represent faithfully the structure used by the VerCors tool-suite to model and specify GCM applications.

The definitions below rely on several predefined structures. *Type* represents a type, we have several kinds of names (even if there is no need to distinguish them strictly): *Name* is an interface name, *CName* and C are component names, *MName* is a method name.

2.3.1. Syntax and Notations

In the following definitions, we extensively use indexed structures (maps) over some countable indexed sets. The indices will usually be integers, bounded or not. Such an indexed family is denoted as follows: $a_i^{i\in I}$ is a family of elements a_i indexed over the set I. Such a family is equivalent to the mapping $(i{\mapsto}a_i)^{i\in I}$. To specify the set over which the structure is indexed, indexed structures are always denoted with an exponent of the form $i \in I$ (arithmetic only appears in the indexes if necessary). Consequently, $a_i^{i\in I}$ defines first I the set over which the family is indexed, and then a_i the elements of the family.

For example $a^{i\in\{3\}}$ is the mapping with a single entry a at index 3; exceptionally, such mappings with only a few entries will also be denoted $(3{\mapsto}a)$. When this is not ambiguous, we shall use abusive vocabulary and notations for sets, and typically write "indexed set over I" when formally we should speak of multisets, and "$x \in A_i^{i\in I}$" to mean $\exists i \in I.\, x = A_i$. An empty family is denoted $[]$. To simplify equations, an indexed set can be denoted $\overline{M}$ instead of $M_l^{l\in L}$ when L is irrelevant.

Let $\uplus$ be the disjoint union operator on indexed sets. The elements of the union are then accessed by using an index of one of the two joined families. We suppose here that disjoint unions are always well defined; this requires to choose disjoint indices in the definition of the component system (e.g. name of methods of different interfaces).

$\{\!\{y \leftarrow x\}\!\}$ is the substitution operation that replaces in a term all occurrences of the term y by the term x (note that the absence of binders makes this operation trivial).

2.3.2. Interfaces

Let *SItf* be the description of a server interface, it is characterised by the interface name, the interface cardinality and the signature of one or more service methods. In the same way, *CItf* is the description of a client interface, containing one or more client methods. We use *Itf* to range over interfaces that can be either client or server ones. Interfaces are structural entities that will be used to attach component binding and thus to specify the flow of messages exchanged by the components. This is why their names are crucial: the component bindings will refer to interface names. Interfaces (also called ports) ensure re-usability and compositionality of components; indeed the interface is the only entity referenced both by the business code (that emits and receives operations on

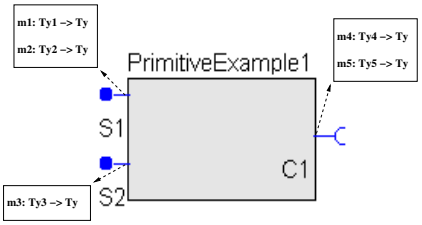

Figure 2: Simple Primitive Component

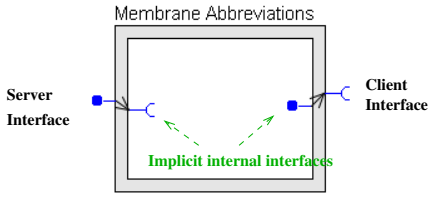

Figure 3: Abbreviations for matching external/internal interfaces

interfaces) and by the composition layer (that binds interfaces together). A method signature, *MSignature*, consists of a method name, an argument type[1], and a return type.

$$SItf \quad ::= \quad (Name, MSignature_i^{i \in I})_S \tag{1}$$

$$CItf \quad ::= \quad (Name, MSignature_i^{i \in I})_C \tag{2}$$

$$Itf \quad ::= \quad SItf \mid CItf \tag{3}$$

$$MSignature \quad ::= \quad MName : Type \rightarrow Type \tag{4}$$

2.3.3. Components

From those definitions we define components, which can be either primitive or composite ones.

$$\begin{aligned} Comp ::= \quad & CName < SItf_i^{i \in I}, CItf_j^{j \in J}, Desc > \\ & \mid CName < SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, Binding_l^{l \in L} > \end{aligned}$$

A *primitive component* consists of a name *CName*, a set of Server Interfaces S_{Itf}, a set of Client Interfaces C_{Itf}, and a description of its behaviour. In the context of this article this description should be sufficient to infer the behaviour of each of the service methods but in practice it would also contain implementation details allowing the instantiation of the component.

Figure 2 illustrates a simple configuration of a primitive component. It exposes two server interfaces. It has also a single client interface showing 2 client methods. The corresponding pNet system will be drawn in Figure 4.

A *composite component* consists of a set of sub-components exporting some server interfaces, some client interfaces, and bindings (see Figure 1). A *binding* connects two interfaces of two components. These can be either two sub-components of a composite, or one is a sub-component and the other is the composite. A binding is thus a pair of qualified names. Each qualified name

[1] it is not restrictive to consider methods with a single argument as we have no restriction on type complexity

is made of two parts, the first one is the name CN of a (sub)component, the second one is the name of an interface IN.

$$Binding \quad ::= \quad (QName,\ QName) \tag{5}$$
$$QName \quad ::= \quad CN.IN \tag{6}$$

We define a function Interfaces that given a component returns the indexed set of its interfaces, and a function Name that returns the name of its argument that can be a component, a method, or an interface.

Membrane and internal interfaces. In Fractal and GCM, the frontier of a composite component is called a membrane and can intercept incoming calls; it deals with the component management. Also, for each interface declared in the composite component definition, a symmetric internal interface is created with the same name and a symmetric role (server or client) as illustrated in Figure 3. In our formalisation, we choose to have no membrane and not specify explicit internal interfaces, more precisely the membrane is trivial, i.e. it has no content and the internal interfaces match exactly the external ones.

2.3.4. Auxiliary functions

We first define an auxiliary function that takes the symmetric of an interface. It takes an interface and returns the same interface with a symmetric role:

$$\text{Symm} : \ Itf \ \rightarrow \ Itf$$

$$\text{Symm}(Name, \overline{MSignature})_S = (Name, \overline{MSignature})_C$$
$$\text{Symm}(Name, \overline{MSignature})_C = (Name, \overline{MSignature})_S$$

Symm is extended to sets of interfaces. We then rely on an auxiliary function GetItf, defined by the following rules, that returns the interface in a composite component that corresponds to a qualified name. If the qualified name is equal to the name of the composite then GetItf returns the inner interface named IN of the composite component; it is the symmetric of the external interface of name IN. Else, CN should correspond to the name of an inner component and GetItf returns the external interface named IN of the inner component named CN. The following rules define the function $GetItf$; this function will be used when we build the behavioural semantics of composite components (see Section 4.2).

$$\text{GetItf} : QName \ \times \ Comp \ \rightarrow \ Itf$$

$$\frac{CN \neq CName}{k \in K \qquad \text{Name}(Comp_k) = CN \qquad Itf \in \text{Interfaces}(Comp_k) \qquad \text{Name}(Itf) = IN}{\text{GetItf}(CN.IN, CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >) = Itf}$$

$$\frac{CN = CName \qquad i \in I \qquad SItf_i = (IN, MSignature_l^{l \in L})_S}{\text{GetItf}(CN.IN, CName < SItf_i^{i \in I}, \overline{CItf}, \overline{Comp}, \overline{Binding} >) = \text{Symm}(SItf_i)}$$

$$\frac{CN = CName \qquad j \in J \qquad CItf_j = (IN, MSignature_l^{l \in L})_C}{\text{GetItf}(CN.IN, CName < \overline{SItf}, CItf_j^{j \in J}, \overline{Comp}, \overline{Binding} >) = \text{Symm}(CItf_j)}$$

2.3.5. Well-formed components

In our semantics, we only deal with components that are correctly formed, for this we define a predicate *WF* that indicates whether a component is well-formed. We suppose there is a sub-typing relation $\trianglelefteq$ between types, and that this relation is classically extended to method signatures, and to families of method signatures.

We first define the predicate UniqueItfNames that takes a set of interfaces and returns true if no two of these interfaces have the same name.

$$\text{UniqueItfNames}(Itf_i^{i \in I}) \Leftrightarrow \left(\forall i, i' \in I.\, i \neq i' \Rightarrow \text{Name}(Itf_i) \neq \text{Name}(Itf_{i'})\right)$$

Then a primitive component is well-formed if all its interfaces have distinct names.

$$WF(CName < \overline{SItf}, \overline{CItf}, Desc >) \Leftrightarrow \text{UniqueItfNames}(\overline{SItf} \cup \overline{CItf})$$

Finally a composite component is well-formed if all its sub-components are well-formed, all the bindings connect existing interfaces of a compatible type[2], all sub-components have distinct names, and no two bindings start from the same client interface. To simplify the semantics, we additionally require that no binding has the same component as source and destination: there is no binding looping back *directly* to the same component. The last condition of the definition is the one specific to our model, it is not generally required by the GCM model[3]. If C has the form $C = CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >$

[2]Those interfaces are found thanks to the GetItf function: they are either interfaces of sub-components or internal interfaces of the composite component.

[3]The reason why we disallow loops is that a loop binding involves a synchronization between emission and reception operations *in the same component* and handling them requires additional rules in the behavioural semantics. For example concerning method invocation, we would need an additional synchronization vector that synchronizes the method body and the request queue of the *same component* in the case of a loop binding (Section 4).

then

$$
WF(C) \Leftrightarrow \left\{
\begin{array}{l}
\text{UniqueItfNames}(\overline{SItf} \uplus \overline{CItf}) \wedge \\
\forall k, k' \in K.\, k \neq k' \Rightarrow \text{Name}(Comp_k) \neq \text{Name}(Comp_{k'}) \wedge \\
\forall k \in K.\, WF(Comp_k) \wedge \\
\forall (Src, Dst) \in \overline{Binding}.\big(\exists Name,\, Name',\, M,\, M',\, \overline{MSignature},\, \overline{MSignature'}. \\
\quad \text{GetItf}(Src, C) = (Name, \overline{MSignature})_C \wedge \\
\quad \text{GetItf}(Dst, C) = (Name', \overline{MSignature'})_S \wedge \\
\quad \overline{MSignature'} \trianglelefteq \overline{MSignature} \wedge \\
\quad \nexists Dst'.\, Dst' \neq Dst \wedge (Src, Dst') \in \overline{Binding}\ \big) \wedge \\
\forall (C_0.CI, C_1.SI) \in \overline{Binding}.\, C_0 \neq C_1
\end{array}
\right.
$$

3. The pNets model: a formalism for defining behavioural semantics

This section provides a "simplified" version of pNets [5] that is convenient for providing a concise formal definition both of pNets themselves, and of the component specification in terms of pNets. The previous definition was more verbose and more complex, but allowed a more efficient implementation and greater expressiveness. In particular, the previous definitions featured a specific construct for pNet families that was more optimised but required additional semantic definitions. In this article, families of pNets will be defined as a special construct that generates a simple pNet (with all the possible synchronization vectors) from a set of pNets. Section 6.3 will show an optimised instantiation of the produced pNet structure and synchronisation vectors.

An operational semantics for pNets is given in Appendix A.

3.1. Term algebra

In all forthcoming definitions, we suppose that we have a fixed set of parameters, used to construct the expressions of our term algebra. Our models rely on the notion of parameterised actions. We leave unspecified the constructors of the algebra that will allow building actions and expressions used in our models. Let us denote Σ the signature of those constructors, and $\mathcal{T}$ be the term algebra of Σ over the set of variables $\mathcal{P}$. We suppose that we are able to distinguish in $\mathcal{T}$ a set of *action terms* (over variables of $\mathcal{P}$) denoted $\mathcal{A}$ (*parameterised actions*), a set of *data expression terms* (disjoint from actions) denoted $\mathcal{E}$, and, among expressions, a set of *boolean expressions* (guards) denoted $\mathcal{B}$. For each term $t \in \mathcal{T}$ we define $fv(t)$ the set of free variables of t.

We also allow countable indexed sets to depend upon variables to allow these sets to have a parameterised size. This allows us to express the semantics of systems of unbounded size while allowing us to consider finite models by defining a value for the parameterised size of each element of the system. We denote $\mathcal{I}_\mathcal{P}$ the set of indexed sets using variables of $\mathcal{P}$. There must exist an inclusion relationship $\subseteq$ over the indexed sets of $\mathcal{I}_\mathcal{P}$, with the natural guarantee that this operation ensures set inclusion when one replaces variables by their values. In practice we will mostly use intervals for which the upper bound depends on the variables of $\mathcal{P}$: $\mathcal{I}_\mathcal{P} = [1..n]$ where n is an integer.

3.2. The pNets model

In this section, we define the structure of pLTSs, pNets and Queues, and define their operational semantics. The formal properties of pNets have been further studied in [29].

A pLTS is a labelled transition system with variables; a pLTS can have guards and assignment of variables on transitions. Variables can be manipulated, defined, or accessed in states, actions, guards, and assignments.

We first identify the actions a pLTS can emit or receive. Let a range over action labels, op over operators, and x over variable names. The set $\mathcal{A}$ of action terms used in pLTSs is defined as follows:

$\alpha \in \mathcal{A}$	$::=$	$a(p_1, \ldots, p_n)$	action terms
p	$::=$	$?x \mid Expr$	action parameters (input or output)
$Expr$	$::=$	$Value \mid x \mid op(Expr_1, .., Expr_n)$	Expressions

For any action term $a(p_1, \ldots, p_n)$, we suppose that each input variable does not appear in any other parameter of the action term: $p_i = ?x \Rightarrow \forall j \neq i.\, x \notin fv(p_j)$.

Actions do not have an input or output role, as each action can emit and receive parameters at the same time. For $\alpha \in \mathcal{A}$ we also suppose that there is a function $iv(\alpha)$ that returns a subset of $fv(\alpha)$ which are the input variables: $iv(a(p_1, \ldots, p_n)) = \{x \in fv(a(p_1, \ldots, p_n)) | \exists i \in [1..n].\, p_i = ?x\}$.

Definition 1 (pLTS). A parameterised LTS is a tuple $pLTS \triangleq \langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle$ where:

- S is a set of states.
- $s_0 \in S$ is the initial state.
- L is the set of labels of the form $\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle$, where $\alpha \in \mathcal{A}$ is a parameterised action, $e_b \in \mathcal{B}$ is a guard, and the variables $x_j \in P$ are assigned the expressions $e_j \in \mathcal{E}$. Variables in $iv(\alpha)$ are assigned by the action, other variables can be assigned by the additional assignments.
- $\rightarrow \subseteq S \times L \times S$ is the transition relation.

The semantics of pLTS is quite standard, they form a labelled transition system with parameters. The state of the system is characterized both by a state in S, and by the value of each variable. A transition fires an action, receiving values for input parameters of the action, and assigning values to some other variables. Guards decide whether a transition is enabled. Note that we make no assumption on finiteness of S or of branching in $\rightarrow$.

pNets are constructors for hierarchical behavioural structures: a pNet is formed of other pNets, or pLTSs at the bottom of the hierarchy tree. Message queues can also appear in leaves of a pNet system. A composite pNet consists of a set of pNets exposing a set of actions, each of them triggering internal actions in each of the sub-pNets. The synchronisation between global actions and internal actions is given by *synchronisation vectors*: a synchronisation vector

synchronises one or several internal actions, and exposes a single resulting global action. Actions involved at the pNet level do not need to distinguish between input and output variables. The set $\mathcal{A}_S$ is the subset of action terms $\mathcal{A}$ that are used in pNets; it only contains actions without input variable:

$$\alpha \in \mathcal{A}_S \quad ::= \quad a(Expr_1, \ldots, Expr_n)$$

Definition 2 (pNets). A pNet is a hierarchical structure where leaves are pLTSs (or queues defined below), and nodes are synchronisation artefacts: $pNet \triangleq pLTS \mid Queue(\overline{m}) \mid \langle\!\langle pNet_i^{i\in I}, SV_k^{k\in K} \rangle\!\rangle$ where

- $I \in \mathcal{I}$ is the set over which sub-pNets are indexed, $I \neq \emptyset$.
- $pNet_i^{i\in I}$ is the family of sub-pNets.
- $SV_k^{k\in K}$ is a set of synchronisation vectors ($K \in \mathcal{I}$). $\forall k \in K, SV_k = \alpha_j^{j\in J_k} \rightarrow \alpha'_k$ where $\alpha_j, \alpha'_k \in \mathcal{A}_S$. Each synchronisation vector verifies: $J_k \in \mathcal{I}$, $\emptyset \neq J_k$, and $J_k \subseteq I$.

For each pNet, we define a function sort ($Sort : pNet \rightarrow \mathcal{A}$). The sort of a pNet is its signature: the set of actions that a pNet can perform. For a pLTS we do not need to distinguish input variables. More formally:

$$\text{Sort}(\langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle) = \{\alpha\{\!\{?x \leftarrow x | x \in iv(\alpha)\}\!\} | \langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J} \rangle \in L\}$$
$$\text{Sort}(\langle\!\langle \overline{pNet}, \overline{SV} \rangle\!\rangle) = \{\alpha'_k |\ \alpha_j^{j\in J_k} \rightarrow \alpha'_k \in \overline{SV}\}$$

The semantics of pNets consists in synchronizing the actions of sub-pNets according to the synchronisation vectors. A pNet can evolve and fire an action only if a synchronisation vector exists for this action and all the sub-pNets involved in the synchronisation vector can perform the action specified for them. In other words, a synchronisation vector SV_k of the form $\alpha_j^{j\in J_k} \rightarrow \alpha'_k$ means that if each sub-pNet j in J_k performs synchronously the action α_j; this results in a global action labelled α'_k. For example, the synchronisation of the same action in two processes indexed i and j corresponds to the synchronisation vector $(i{\mapsto}a, j{\mapsto}a) {\rightarrow} a'$ (recall that we identify indexed sets and mappings, giving us a convenient notation for synchronisation vectors).

When $I = [1..n]$, it is equivalent to use tuple notations instead of indexed sets. In that case, we denote the pNet as $\langle\!\langle pNet_1, \ldots, pNet_n, SV \rangle\!\rangle$, and each synchronisation vector as: $\langle \alpha_1, \ldots, \alpha_n \rangle \rightarrow \alpha$. In that case, elements not taking part in the synchronisation are denoted by a dash $'-'$ as in: $\langle -, -, \alpha, -, - \rangle {\rightarrow} \alpha$.

Flow of information. The synchronisation vectors allow a lot of flexibility in the use of the variables, and the decidability of whether a synchronisation vector can be triggered depends a lot on the precise action algebra (operators, allowed expressions,...). In this article, however, this expressive power is not used: for each globally synchronised action there is a single pLTS that outputs the value of each parameter (and potentially several targets). Several actions will receive one parameter and output another one, without any consequence on the decidability

of which actions can be triggered. In the drawings, arrows will always indicate the direction of the flow of information (however sometimes, this flow is not unidirectional, in that case we indicate the most obvious direction and explain the details in the text). By convention, when we need to distinguish several action names, we will precede the actions that mostly have an input role with an i prefix, like iQ and iR.

Queues. We also define a particular pNet called $Queue(\overline{m})$; it models the behaviour of a FIFO queue, with $\overline{m}$ the set of enqueue-able elements.We suppose that the term algebra has two specific constructors Q and $Serve$[4] such that[5]

$$\forall m_i \in \overline{m}.\, Serve\_m_i \in \mathcal{A}_S \wedge iQ\_m_i \in \mathcal{A}_S$$

Then the queue pNet offers the following actions: $L = \{iQ\_m_i | m_i \in \overline{m}\} \cup \{Serve\_m_i | m_i \in \overline{m}\}$. The behaviour of a queue is only FIFO en-queueing (denoted iQ_m_i) and de-queueing (denoted $Serve\_m_i$) of messages. It could be encoded as an infinite pLTS.

$$\mathrm{Sort}(Queue(\overline{m})) = \{iQ\_m_i | m_i \in \overline{m}\} \cup \{Serve\_m_i | m_i \in \overline{m}\}$$

Whenever pNets will be translated into (ultimately finite) automata structures for model-checking, pNet queues will naturally be represented by finite automata. However, in order to be able to address more general approaches, and in particular specific model-checking algorithms for unbounded channels, we keep a high-level representation of queues. From these abstract queues, we will be able to generate both regular representation (for unbounded queues), and finite representation (for explicit-state model-checking).

3.3. Additional pNet Constructs

pNet families. It is convenient to collect a set of pNets into a family that can be addressed according to an index. For example such a construct makes it easy to model a pool of identical proxies responsible for handling futures; families are also convenient to gather the pNets of all the subcomponents of a given component into a single sub-pNet (that represents the functional content of the component). We define a construct that builds families of pNets as syntactic sugar. This is simpler than adding families to the pNet definition and defining a semantics for them.

We define a constructor for a pNet made of an indexed family of pNets. $\langle\!\langle \overleftrightarrow{PN_i^{i \in I}} \rangle\!\rangle$ takes a family of pNets indexed over a set $I \in \mathcal{I}$ and produces a global pNet. The synchronisation vectors for this family will be expressed at the level

[4] We chose constructors coherent with the term algebra we will use in this article to simplify notations.

[5] In the rest of the paper, most action names contain names of methods. This way the name of the method involved in the interaction is exhibited in the action label.

above, consequently we "export" all the possible synchronisation vectors that the family could offer, only some of them will be used.

$$\langle\!\langle \overleftrightarrow{PN_i^{i\in I}} \rangle\!\rangle \triangleq \langle\!\langle PN_i^{i\in I}, \{\overline{\alpha} \to \overline{\alpha} | \overline{\alpha} \in V\} \rangle\!\rangle \quad \text{where } V = \{\alpha_j^{j\in J} |\, J \subseteq I \wedge \forall j \in J.\, \alpha_j \in \mathrm{Sort}(PN_j)\}$$

This supposes that the elements of V are action terms. If all the elements of the family are identical, then we simply write $\langle\!\langle \overleftrightarrow{PN^I} \rangle\!\rangle$.

In order to express synchronisation vectors of families of pNets, we must allow families of actions to be considered as actions themselves. More precisely, if a_i is an action, then actions can be of the form $(a_i)^{i\in I}$, or $i{\mapsto}a$ to allow the sub-pNet at index i to perform an action.

Synchronised Actions. Finally, we identify the actions that are already synchronised (they will not need further synchronisation). We slightly extend the action algebra with such already synchronised actions (distinguished by underlined labels):

$$\alpha \in \mathcal{A}_S \quad ::= \quad a(Expr_1, \ldots, Expr_n) \mid \underline{a}(Expr_1, \ldots, Expr_n) \quad \text{pNet actions}$$

Synchronised actions are not meant to be used any more for synchronisation purposes, they should just be visible at the top-level of the pNet hierarchy. Note that action labels that have an input connotation like iQ and iR will not appear underlined.

We define an operator that takes an indexed set of pNets and returns the synchronisation vectors that should be included in the parent pNets to allow the visibility of synchronised actions:

$$\mathrm{TauMonitor}(pNet_i^{i\in I}) = \{(i{\mapsto}\underline{\alpha}) \to \underline{\alpha} | i \in I \wedge \underline{\alpha} \in \mathrm{Sort}(pNet_i)\}$$

In the following, those synchronisation vectors dedicated to observation will be *implicitly* included as synchronisation vectors of all our pNets. This means that for all pNets, $\mathrm{TauMonitor}(pNet_i^{i\in I})$ is implicitly included in the set of synchronisation vectors (where $pNet_i^{i\in I}$ is the set of sub-pNets of the new pNet).

In this article, we do not use explicitly invisible actions (τ actions). More precisely, τ actions would behave similarly to the synchronised actions defined above: there is always a synchronisation vector allowing a sub-pNet to perform a τ action. Also, synchronized actions exported by the TauMonitor operator are only useful to observe the internal reductions. To reduce the size of the model one could replace them with a τ action without any consequence on the semantics.

3.4. Adequacy of pNets for modelling GCM components

Next sections will present the behavioural semantics of GCM components, expressed as a translation from a component architecture into a hierarchy pNet.

Before defining this translation, we explain below why we chose pNets as a support for GCM behavioural semantics.

First, as already explained, our goal is to provide a model adapted to the behavioural verification of the properties of GCM applications. It must be adapted to the generation of a model that can then be verified, typically a finite model that could be model-checked, even if other techniques could be envisioned. In pNets, models can use parameters, both in the structure and in the LTSs which allows us to give a semantics based on an infinite set of states, but also to easily consider finite instances by restricting each parameter to a finite domain. Thus the first reason for the choice of pNet is that *it is adapted to the definition of infinite models from which a finite instance can easily be extracted.* Defining finite instances in this manner can be done using abstract interpretation techniques, while preserving logical properties of the system [30].

Second, concerning communication, the semantics of communication and asynchrony of pNets fits closely to the one of GCM. Indeed, to guarantee causal ordering of requests, GCM components communicate by a rendez-vous mechanism. In GCM/ProActive the request sending and its arrival in the queue of the destination component occur synchronously. The rest of the execution is entirely asynchronous. pNets have a similar semantics: they are made of independent pLTSs or pNets interacting by synchronous communications. On the contrary, futures are a too high-level construct to be part of pNet definition. They will have to be encoded by a set of pLTSs. Next sections will show that the parameterised nature of pNets and the synchronisation vectors allow for encoding futures in a precise and generic way. Thus the second reason why we chose pNet is that *it provides a communication model similar to GCM for requests and give enough expressive power to encode futures.*

From another point of view, we mentioned earlier that we want our behavioural models to represent the structure of the application (e.g. to allow the encoding of reconfigurations). On that aspect and more generally when encoding communication channels, π-calculus might seem to be a reasonable approach. However, we think that channels *à la* π-calculus are too powerful for automatic verification techniques. Indeed, in GCM/ProActive bindings are not first-class entities and can only be reconfigured by an application manager. Additionally, pNets are much better adapted to the verification techniques we target, i.e. finite state model-checking, than π-calculus.

Finally, *the hierarchical structure of pNets fits well with hierarchical components.* The hierarchical structure of the ADL and of pNets will be the same, even if additional pLTSs and pNets will be defined to encode specific features (e.g., future proxies).

Next section will illustrate precisely how pNets provide a convenient abstraction for modelling asynchronous components communicating by asynchronous requests and futures.

4. Behavioural semantics for GCM components

This section defines formally the behavioural semantics for the component model defined in Section 2.3. It shows how to build pNets from the specification of a hierarchy of components. We organise this section as follows. We first give a behavioural semantics for primitive components including simple future proxies, and the behaviour and synchronisation of the different elements of the primitive component. We then describe the behavioural semantics for composite components, which compose the semantics of their sub-components synchronising the request and replies among the sub-components, the composite, and the external components. For that, we will need to define a new kind of future proxies for handling the delegation mechanism that occurs in the composite components.

Term algebra. The term algebra we use is a set of parameterised actions; actions will typically be of the form *Serve_m* for m a method label as defined below. Parameters will be either values (method parameters denoted by *arg* or computed results denoted *val*), or future identifiers (denoted by either p, or f, or *fid*). Throughout this article, we use, as action parameters, two variables *arg* and *val* that (implicitly) range over the set of *values*, this set of values being purposely undefined. In an object-oriented language, those values should be an abstraction of objects.

Labels for identifying methods. In our actions, we need identifiers for methods that are more precise than simple method names. We define thus *MethodLabels* as a set of method labels, where a method label encompasses a method name, a signature, and the interface the method belongs to, plus possibly other meta-information. Most of the following can be read as if *MethodLabels* were just method names, however at some specific points and to disambiguate different methods, the other information encoded in *MethodLabels* is also necessary. m_i range over such method labels.

A function MethLabel : $Itf \rightarrow \mathcal{P}(MethodLabels)$ is defined, where MethLabel(Itf_i) returns the set of *MethodLabels* corresponding to the methods of interface Itf_i. MethLabel is also defined for sets of interfaces (union of method labels for each interface). Conversely, for a given method label m, $Itf(m)$ returns the interface of the method.

Behavioural semantics. The behavioural semantics of components is expressed under the form $[\![Component]\!]$[6]. It relies on several auxiliary functions for expressing the semantics of specific parts of the components. The core entities of this semantics include: a *queue* accepting incoming requests, a *body* entity that serves requests and triggers the adequate service methods, *future proxies* that act as place-holder for awaited request results, and *proxyManager*s that manage a pool of future proxies.

[6] Appendix B provides the list of semantic definitions contained in this article with their brief description and their signature.

4.1. Semantics of primitive components

Primitive components are the leaves of the hierarchy; they contain the applicative code from which more complex components, and thus more complex behaviours can be built. This section gives a behavioural semantics for GCM primitive components, able to receive requests, to serve them in a FIFO order by executing a service method, and to send requests to the external world. Additionally to the global structure of a primitive component and the synchronisation of its sub-entities, this section defines pLTSs describing the behaviour of a FIFO service policy, of proxies for handling futures, and of managers for pools of future proxies.

The general idea used here is that the structure of the pNets encoding a component reflects as closely as possible the implementation provided by the ProActive/GCM middleware. This includes the body, the queue, and the future proxies. The default behaviour of bodies and queues is to serve requests, one at a time, in a FIFO order, and we encode this straightforwardly in the pNets. The future proxies, in the implementation, are allocated dynamically, and ultimately managed by the garbage collector, while in the static pNet model, we encode them as an indexed family, and provide an explicit recycle operation allowing to reduce the size of the model for model-checking.

4.1.1. Illustrative Example

We first illustrate and explain the structure of the behavioural semantics of primitive components based on the component shown in Figure 2. Figure 4 illustrates the structure of the pNet expressing the semantics of the component. It illustrates both the global structure of the pNets represented by boxes, and the synchronisation vectors represented by arrows (an ellipse is used when a synchronisation vector involves more than two processes), and labelled by their corresponding rule number. The inference rules will be defined in Table 1, and explained in Section 4.1.6. Note that the direction arrows is purely conventional, but goes, as much as possible, from an emission action to a reception action, intuitively following the data flow. Arrows representing complete vectors in the figure are labelled by the corresponding *synchronized action* (underlined), while those at the outside border of the pNet show only the corresponding incoming or outgoing action (not yet synchronized).

A primitive component can receive incoming requests (iQ_m_i) that are stored in the *Queue* pNet and then served by the *Body* pLTS. The queue allows the component to always be ready to accept the new request while being able to handle the service of the request asynchronously. The service consists in triggering a $Call\_m_i$ to the adequate service method, called $\mathcal{M}i$ in the figure. The service method should ultimately produce a result for the request. Once a result is computed for the request, an R_m_i action is emitted with the right future identifier *fid* and result value *val*.

The service method can call external components through client interfaces. For each method of each interface there is a proxy manager PM_*, in charge of creating and managing a family of future proxies $Proxy\_*[< proxy - index >]$.

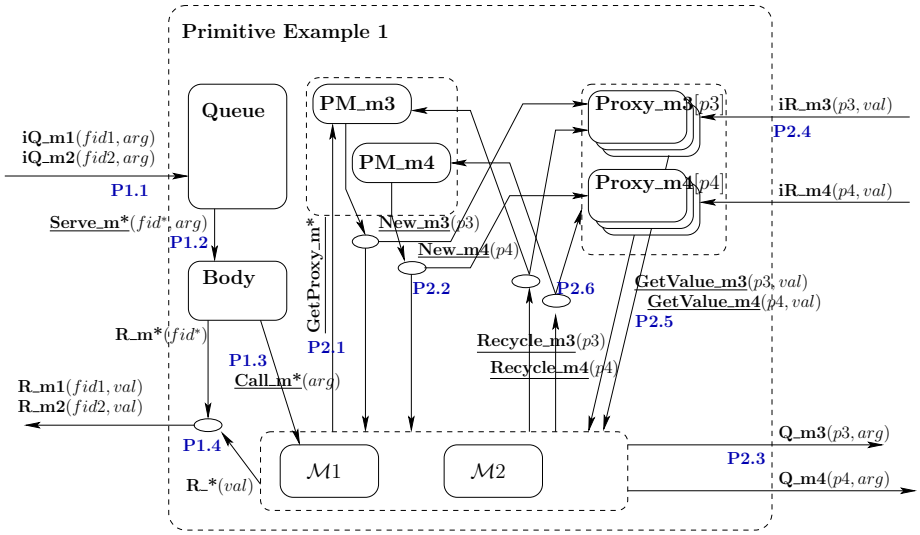

Figure 4: pNet for the Simple Primitive Component from Figure 2

Those future proxies have two purposes: they will be accessed when the result of the request is needed, and they will be given the computed result when the invoked request is finished. Each future inside a proxy family keeps track of one specific remote method invocation, and because of asynchrony there can be several instances in the same family that are active at the same time. On each proxy manager PM_m_i, the caller can perform a *GetProxy*. Upon request, a fresh future proxy $Proxy\_m_i$ is allocated and returned by a New_m_i action that acts as a response to the *GetProxy*. Then the outgoing call is emitted with the reference to the corresponding proxy sent as parameter (Q_m_i). Later, when a result is computed, the reply iR_m_i is received by the adequate proxy and the result can be accessed by $GetValue\_m_i$ actions performed by some service methods. Then, service methods can emit *Recycle* actions that are sent to the adequate proxy and proxy manager. Proxies are indexed in the same way as the default ProActive implementation (alternative indexing mechanisms exits in the library): only a local index is created allowing for simple creation of small future identifiers; then the transmission of the future value follows the invocation flow (i.e. the bindings) backwards. This way the structure for future indexes is small and simple, and future transmission is done locally, without needing a global referencing system, and following a single binding. Section 5 will show that, when futures can be passed between components, the proxy and index need to be more complex. Figure 14 shows an example of a service method that creates and uses several future proxies, it illustrates one possible workflow involving the actions described above.

4.1.2. pNets

This section formalises and generalises the principles depicted in the previous section. Primitive components encode the business code of the application. Consequently, the behaviour of primitive components include the behaviour of "service methods": those methods represent the business logic of the application. This behaviour is the only part for which we do not specify the generation of the behavioural model, it can be inferred by static analysis or written by the programmer. We suppose that for each service method m_l, $[\![m_l, Desc]\!]_{service}$ provides a pNet expressing the behaviour of this service method. Concerning the rest of the behaviour of a primitive component below, it is computed by the rules shown in this section.[7]

$$\frac{\begin{array}{c} m_l^{l\in L} = \text{MethLabel}(\overline{SItf}) \quad \mathcal{Q} = Queue(m_l^{l\in L}) \quad \mathcal{B} = [\![m_l^{l\in L}]\!]_{body} \\ \forall l \in L.\, \mathcal{SM}_l = [\![m_l, Desc]\!]_{service} \qquad \forall j \in J.\mathcal{P}_j = \mathcal{P}(CItf_j) \\ \forall j \in J.\mathcal{PM}_j = \mathcal{PM}(CItf_j) \qquad SV = SV_S(m_l^{l\in L}) \cup SV_C(CItf_j^{j\in J}, L) \end{array}}{[\![CName < \overline{SItf}, CItf_j^{j\in J}, Desc >]\!] = \langle\!\langle \mathcal{Q}, \mathcal{B}, \langle\!\langle \overleftrightarrow{\mathcal{SM}_l^{l\in L}} \rangle\!\rangle, \langle\!\langle \overleftrightarrow{\mathcal{PM}_j^{j\in J}} \rangle\!\rangle, \langle\!\langle \overleftrightarrow{\mathcal{P}_j^{j\in J}} \rangle\!\rangle, SV \rangle\!\rangle}$$

With the auxiliary rules:

$$\frac{m_n^{n\in N} = \text{MethLabel}(CItf) \qquad \forall n \in N.\, \mathcal{F}_n = \langle\!\langle \overleftrightarrow{[\![m_n]\!]^{\mathbb{N}}_{proxy}} \rangle\!\rangle}{\mathcal{P}(CItf) = \langle\!\langle \overleftrightarrow{\mathcal{F}_n^{n\in N}} \rangle\!\rangle}$$

$$\frac{m_n^{n\in N} = \text{MethLabel}(CItf)}{\mathcal{PM}(CItf) = \langle\!\langle \overleftrightarrow{[\![m_n]\!]^{n\in N}_{proxyManager}} \rangle\!\rangle}$$

Each premiss of this rule correspond to one item in the following list:

- A queue able to receive incoming requests: it can enqueue a request on a method label of one of the server interfaces, we use here a pNet queue constructor.
- A body (see Section 4.1.3) that will serve all the requests present in the queue. The body takes one request after the other, delegates the treatment of the request to the service methods $\langle\!\langle \overleftrightarrow{\mathcal{SM}_l^{l\in L}} \rangle\!\rangle$, and only serves the next request when the previous one is finished.
- Service methods: there is one service method for each method label of a server interface. The service methods provide the business logic of the primitive components.
- A family $\mathcal{PM}$ of proxy managers (see Section 4.1.5) indexed both over the set of client interfaces and over the methods of those interfaces: those

[7] Recall that the construct $m_l^{l\in L} = \text{MethLabel}(SItf_i^{i\in I})$ defines both the value of each method label m_l and the set L over which it is indexed.

managers are responsible for allocating a new proxy when requested, and activating those newly created proxies.

- A family $\mathcal{P}$ of future proxies (see Section 4.1.5) indexed over the set of client interfaces (J), the methods of those interfaces (N), and proxy indexes, i.e. integers ($\mathbb{N}$): a proxy is responsible for receiving the result of a request made towards another component; when the value of the result is needed by a service method, this method asks for the value to the adequate proxy.

4.1.3. Body

The body is a pLTS modelling the service of the different requests: for each service method, the body can dequeue a request corresponding to this method, delegate the service to the appropriate service method, and synchronise with the end of a service method. It waits for the end of one service method before de-queueing a new request. It can be generated automatically from the set of service methods $m_i^{i\in I}$. $[\![m_i^{i\in I}]\!]_{body} = \langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle$ where:

- $S = \{s_0\} \cup \bigcup_{i\in I}\{s_i(fid, arg)\} \cup \bigcup_{i\in I}\{s'_i(fid)\}$
- $L = \bigcup_{i\in I}\{Serve\_m_i(?fid, ?arg), Call\_m_i(arg), R\_m_i(fid)\}$
- $\rightarrow = \{s_0 \xrightarrow{Serve\_m_i(?fid,?arg)} s_i(fid, arg) \big| i \in I\} \cup$
  $\{s_i(fid, arg) \xrightarrow{Call\_m_i(arg)} s'_i(fid) \big| i \in I\} \cup \{s'_i(fid) \xrightarrow{R\_m_i(fid)} s_0 \big| i \in I\}$

For each method m, the body pLTS can always perform the three actions *Serve_m*, then *Call_m* and then *R_m*. This body encodes a *mono-threaded* component behaviour where no two requests are served at the same time. This corresponds indeed to the behaviour of the GCM/ProActive framework, and more generally to the behaviour of active objects or actors. Allowing the body to serve multiple requests at the same time would be quite easy but the resulting behaviour would be much more complex. Figure 5 provides a graphical representation for the pLTS of the body defined above (in the rest of this article we will express pLTSs graphically). The graph shows a body able to serve three functional requests (m_0, m_1, and m_2).

4.1.4. Service Methods

The behaviour for each service method is expressed by a pNet, used when serving the corresponding request. This behaviour is either obtained by source code analysis, or provided by the user. It can for example be composed of the execution of several pLTSs expressing the behaviour of each local method. In the Vercors environment, it can be generated from UML state machine [31].

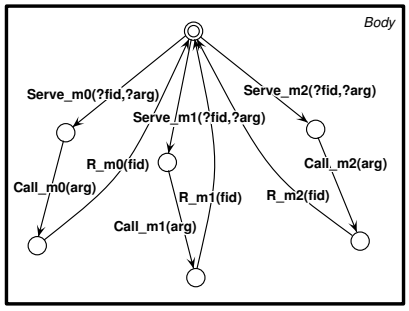

Figure 5: Graphical representation of the behaviour of the Body

4.1.5. Modelling of the Future Proxies

Futures enable asynchronous method invocations. Technically, a future is often implemented by a proxy that represents the result and is accessible both locally to know whether the result came back, and remotely by the invoked component that wants to return the result. We represent those notions in our behavioural models. Fresh future proxies are instantiated upon need; this is done by invoking the proxy manager before performing an asynchronous request.

Remember future proxies are families indexed by client interface index, method index, and future identifier; proxy managers are indexed by client interface index and method index. We provide the specification of proxy manager and future proxy in Figure 6. The behavioural semantics of the proxy manager is defined by the pLTS *ProxyManager_m* shown on the right side of Figure 6; it is denoted by $[\![m]\!]_{proxyManager}$. It maintains a list of available proxies and returns a fresh future (by a *New* action), or if there are no more fresh futures, raises an error *NoMoreProxy*. Indeed, in our specification, we let future identifiers be indexed by $\mathbb{N}$ but if one wants to perform finite model-checking, a bound should be chosen on the size of each future proxy family, and in each proxy manager, *Max_Proxy* should be set to the chosen bound. The proxy semantics we propose here faithfully fits the semantics of GCM/ProActive implementation in case the bound *Max_Proxy* is infinite.

When model-checking a GCM application, we have to chose specific sizes for the proxy families (and similarly for request queues), and we check reachability of the *NoMoreProxy* error. If it is reachable this means that our model does not represent correctly the semantics of the implementation, and we should increase the proxy number. If it is not reachable, we can deduce that it would also be the case for any larger size, and naturally for the infinite case. Proving this cannot be done by model-checking techniques: it should be done inductively, and this is straightforward using the structure of the proxy manager automaton. Whatever the state of the *Pool_Proxy* array is, and considering any possible interaction sequence with the context; when you start in the initial state, set the p index to 0, and go a number of times around the central loop of the automaton searching for a free proxy, then the guard *[p=Max_Proxy]* will necessarily occur later when

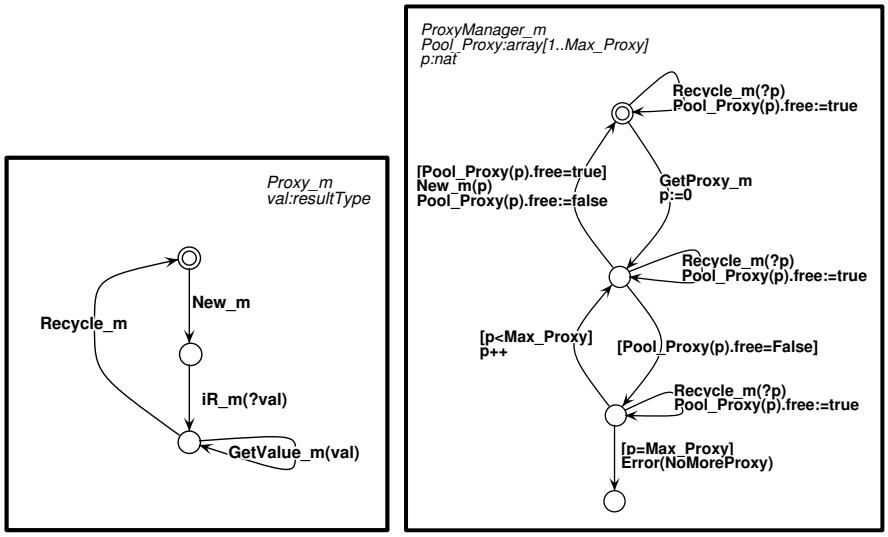

Figure 6: pLTSs for the Future Proxies and Proxy Managers

Max_Proxy is larger.

Each future proxy has a very simple behaviour; it encodes a single assignment memory location; $[\![m]\!]_{proxy}$ is defined by the pLTS *Proxy_m* shown on the left side of Figure 6. Once activated by a *New_m* action, it waits for the corresponding reply ($R\_m(?val)$). At this point, the proxy can be accessed to know the result of the request invocation, it repeatedly sends the result to the service methods by a $GetValue\_m(val)$ action.

The proxies in Figure 6 are endowed with a *Recycle_m* transition, bringing back the proxy in its initial state. This is useful when information can be computed, e.g., by static analysis, that the proxy is not useful any more, so it can be made available again in the proxy pool of the ProxyManager. The *Recycle_m* event should be sent by the LTS modelling a service method. When such an event is received by the ProxyManager, this sets the corresponding entry in the *Pool_Proxy* to *free*.

Several design choices have been made in the specification of proxies and proxy managers. Alternative proxy family specifications, better optimised for some specific usage could also be designed, but should provide a behaviour equivalent to this one. For generality, the proxy manager accepts *Recycle_m* transitions in every state, even if, as a single request is served at a time, this action can only be received when the proxy manager is in its initial state.

4.1.6. Synchronisation Vectors

The set of synchronisation vectors for a primitive component is built (in the semantic rule from section 4.1.2) by two functions: SV_S that provides the set of synchronisation vectors corresponding to the server interfaces, and SV_C for

Table 1: Server and client-side synchronisation vectors for primitive components. Recall that the synchronised sub-pNets (defined in Section 4.1.2) are:
$\langle\!\langle Queue, Body, ServiceMethods, ProxyManagers, Proxies \rangle\!\rangle$

$$\frac{l \in L \qquad fid \in \mathbb{N}}{\begin{array}{ll}\{\langle iQ\_m_l(fid, arg), -, -, -, - \rangle \rightarrow iQ\_m_l(fid, arg), & [1]\\ \langle Serve\_m_l(fid, arg), Serve\_m_l(fid, arg), -, -, - \rangle \rightarrow \underline{Serve\_m_l}(fid, arg), & [2]\\ \langle -, Call\_m_l(arg), l{\mapsto}Call\_m_l(arg), -, - \rangle \rightarrow \underline{Call\_m_l}(arg), & [3]\\ \langle -, R\_m_l(fid), l{\mapsto}R\_m_l(val), -, - \rangle \rightarrow R\_m_l(fid, val)\} & [4]\\ \subseteq SV_S(m_l^{l \in L}) & \end{array}} \text{ P1}$$

$$\frac{j \in J \qquad l \in L \qquad I = \text{MethLabel}(CItf_j) \qquad m_i \in I \qquad p \in \mathbb{N}}{\begin{array}{ll}\{\langle -, -, l{\mapsto}GetProxy\_m_i, j{\mapsto}i{\mapsto}GetProxy\_m_i, - \rangle \rightarrow \underline{GetProxy\_m_i}, & [1]\\ \langle -, -, l{\mapsto}New\_m_i(p), j{\mapsto}i{\mapsto}New\_m_i(p), j{\mapsto}i{\mapsto}p{\mapsto}New\_m_i \rangle \rightarrow \underline{New\_m_i}(p), & [2]\\ \langle -, -, l{\mapsto}Q\_m_i(p, arg), -, - \rangle \rightarrow Q\_m_i(p, arg), & [3]\\ \langle -, -, -, -, j{\mapsto}i{\mapsto}p{\mapsto}iR\_m_i(val) \rangle \rightarrow iR\_m_i(p, val), & [4]\\ \langle -, -, l{\mapsto}GetValue\_m_i(p, val), -, j{\mapsto}i{\mapsto}p{\mapsto}GetValue\_m_i(val) \rangle \rightarrow \underline{GetValue\_m_i}(p, val), & [5]\\ \langle -, -, l{\mapsto}Recycle\_m_i(p), j{\mapsto}i{\mapsto}Recycle\_m_i(p), j{\mapsto}i{\mapsto}p{\mapsto}Recycle\_m_i \rangle \rightarrow \underline{Recycle\_m_i}(p)\ \} & [6]\\ \subseteq SV_C(CItf_j^{j \in J}, L) & \end{array}} \text{ P2}$$

the client interfaces. Each of those sets is defined as the smallest set verifying the constraints given in Table 1.

Let us explain briefly what are the synchronisation vectors generated by the inference rules, more precisely, we focus on the synchronisation vectors for the $GetProxy\_m_i$ actions, Rule [P2.1]. Recall that we are in the context of the rule given in section 4.1.2, which defines L the set of all service methods and J the set of client interfaces. Then rule [P2] defines I as the set of methods in the client interface $CItf_j$. One synchronisation vector for $GetProxy\_m_i$ is generated for each $l \in L$, for each $j \in J$, and for each $i \in I$. Each synchronisation vector synchronises one action $l{\mapsto}GetProxy\_m_i$ of the sub-pNet containing the family of service methods, with one action[8] $j{\mapsto}i{\mapsto}GetProxy\_m_i$ of the sub-pNet containing the family of proxy managers (for each interface). As each of the synchronisation vectors of families of pNets triggers the action on the indexed element of the family, this line allows one action $GetProxy\_m_i$ of one service method (indexed by l) to be synchronised with the action $GetProxy\_m_i$ of the proxy manager indexed by i of the interface indexed by j.

The set of service synchronisation vectors SV_S defined in rule [P1] encodes the following synchronisations: en-queueing an incoming request [P1.1], service of a request by the body [P1.2], the body calling a service method to serve a request [P1.3], and the service method providing a result for this served request [P1.4]. In the last case the result both notifies the body process and is returned

[8] $j{\mapsto}i{\mapsto}a$ should be read $j{\mapsto}(i{\mapsto}a)$

to the outside of the primitive component. In all the actions, the method argument or the returned value is used as parameter, plus when necessary the identifier of the concerned future (*fid*).

The set of client synchronisation vectors SV_C is defined in rule [P2][9]; it encodes the following synchronisations:

- obtaining a new future proxy which involves a call to the proxy manager [P2.1] and another action [P2.2] for returning a fresh proxy identifier and activating the corresponding future proxy;
- the sending of a request from a service method to an external component [P2.3];
- the reception of a result by the future proxy [P2.4];
- the access to a future value [P2.5] from a service method, the future value is stored in the future proxy; it is interesting to note here that the value of p is provided by the *GetValue* action, it is used to index the right future proxy, and the value of *val* is on the contrary provided by this future proxy and "returned instantly" to the service method;
- the eventual recycling of a future proxy [P2.6].

The function SV_C receives as argument the set L of indexes over which service methods range. This argument is necessary because all service methods can perform some of the actions, like $GetValue\_m_i$.

4.2. Semantics of composite components

Hierarchical component models, like GCM, allow the specification of new components, based on the composition of others. Such a compositional approach is very convenient when building large applications. As defined in Section 2.3, we start from a static definition of composition of the system by some form of ADL (architecture description language). The ADL is used to extract component bindings that will define the synchronisation between the emission and the reception of communication actions. A composite component also has a request queue for receiving requests coming from the outside or the inside of the component, it treats each of those requests by sending it to the adequate sub-component or emitting a request to the outside world. To this end, the composite has future proxies but, as the requests only transit through the component, we implement a special future proxy that stores the name of the future that should be fulfilled by the current request. This proxy enables future redirection: when a future proxy is created, it receives the identifier for another future f' and when the reply will come back, it will be immediately re-sent as

[9] Note the indexing of proxy managers (by interfaces and methods) and of proxies (by interfaces, methods, and proxy identifier).

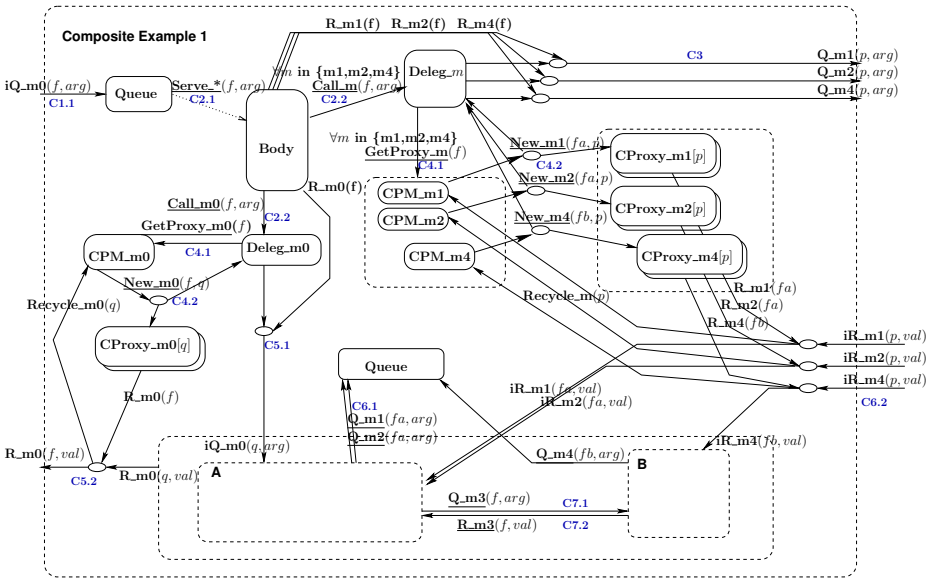

Figure 7: pNet for the Composite Component from Figure 1 For readability we draw the Queue pNet at two places.

a reply for the future f'. The behavioural semantics of the composite component not only encodes the handling of requests and redirection of futures at the boundary of the composition but also encodes the routing of requests in the composite component. This second aspect is different from the implementation where, for efficiency reasons the composite has no control on the communication between its direct sub-components. Here we take a more compositional approach where the composite component controls the communication between its direct sub-entities.

Except for this difference the component structure is similar to the implementation: in ProActive/GCM composite components are implemented by active objects with predefined behavior consisting in delegating the requests they receive to the right target components. They store futures just to transmit their return value to the original caller.

4.2.1. Illustrative Example

Figure 7 shows the pNets structure corresponding to the composite component of Figure 1. It illustrates the structure of the pNets we generate for specifying the behaviour of a composite component. We use it here to illustrate how we are able to generate behavioural models for GCM composites. As for the primitive case, the corresponding semantic rule and vector table will be detailed later, in Section 4.2.2 and Table 2.

Two sub-pNets A and B represent the behaviour of sub-components A and B. A queue pNet receives *iQ_m0(f,arg)* requests where f is the future corresponding

to the request and *arg* the value passed as argument. *Serve_∗* communications allow the body to retrieve those requests, which will then be treated by the *Deleg_m0* pNet, this pNet receives *Call* communications from the body and delegates the request to an inner component (here, A); during this process, a future proxy is created by the proxy manager (process *CPM_m0*), the proxy (process *CProxy_m0[q]*) is responsible for receiving the reply when *A* has finished the request treatment and for forwarding this result to the outside of the composite component: *R_m0(q,val)* that becomes *R_m0(f,val)*. Note that this proxy encodes some basic form of future forwarding: the future *q* corresponds to the same result as the future *f*. This future forwarding mechanism will be extended in Section 5.1 to deal with first-class futures.

Similarly, requests emitted by the inner components arrive in the queue (we draw two *Queue* boxes, but they correspond to the same element), they are then delegated to the outside world by a similar mechanism: a *Deleg_m* pNet delegates the call, and creates a future proxy, which will be responsible for sending back the result to the appropriate inner component. Here again the proxy manages the fact that both the future *q* and the future *fa* (or *fb*) represent the same result.

The structure of the proxy-manager (*CPM_∗*) and of the proxy (*PM_∗*) is similar to the primitive case: using double indexes over interfaces and methods. However, here we have proxies both for client interfaces and for server interfaces (because of their corresponding internal client interfaces).

All the communications expressed above, but also the communication channels between the different inner components – requests *Q_m3* and the corresponding replies *R_m3* – correspond to synchronisation vectors of the pNet of the composite. Each box is a pLTS or a family of pLTSs, except inner components that are more complex pNets.

4.2.2. Global structure

The semantics of a composite component is described below; the first difference compared to the semantics of primitive components is that it does not rely on service method specification, instead it delegates requests to sub-components, some of the sub-pNets of a composite component's pNet correspond to the behaviour of the sub-components. Like primitive components, the pNet corresponding to a composite component comprises a proxy manager and proxy families, but in the case of composite, we also need one future proxy family for each method of each *server interface*. The request queue can receive requests on all methods of all the interfaces of the composite component, both server and client (i.e. internal server) ones. For each declared interface of the composite component, a symmetric (the symmetric of a server interface is a client one) internal one exists, it can be obtained by using the Symm function. The semantics of delegation consists in re-emitting each received request to the symmetric interface, and in using the bindings to route this request to the right target.

Both server and client interfaces play the same role: they transmit both requests and replies. Consequently, they both have delegation methods and future proxies. The transmission mechanism works as follows. To delegate a request

to an inner component or from an inner component to an external one, "delegation methods" are used, they are denoted $\mathcal{DM}$. Delegation methods transform a request into another and a special proxy for future is used to remember the relationship between the original future and the future of the new delegated request. The building of proxy manager families is identical to the case of the primitive component (we reuse the same function); however each future proxy is slightly different as shown below. The body and the queue are similar to the ones of primitive components.

The rule below has a structure very similar to the rule for primitives, the main differences coming from the role of delegation methods, detailed in the next two subsections, and the handling of internal bindings (see Table 2).

$$\frac{\begin{array}{l}\overline{m} = \biguplus_{Itf \in \overline{SItf}} \mathrm{MethLabel}(Itf) \uplus \biguplus_{Itf \in \overline{CItf}} \mathrm{MethLabel}(Itf) \qquad \mathcal{Q} = Queue(\overline{m}) \qquad \mathcal{B} = [\![\overline{m}]\!]_{body} \\ Itf_h^{h \in H} = \overline{CItf} \uplus \mathrm{Symm}(\overline{SItf}) \qquad \forall h \in H.\, \mathcal{P}_h = \mathcal{P}(Itf_h) \qquad \forall h \in H.\, \mathcal{PM}_h = \mathcal{PM}(Itf_h) \\ SV = \; SV_S(\mathrm{MethLabel}(\overline{SItf}), \mathrm{MethLabel}(\overline{CItf})) \; \cup \; SV_C(\overline{CItf}, Itf_h^{h \in H}) \\ \qquad \cup \; SV_B(CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >)\end{array}}{\begin{array}{l}[\![CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >]\!] = \\ \qquad \langle\!\langle \mathcal{Q}, \mathcal{B}, \mathcal{DMS}(\overline{m}), \langle\!\langle \overleftrightarrow{\mathcal{PM}_h^{h \in H}} \rangle\!\rangle, \langle\!\langle \overleftrightarrow{\mathcal{P}_h^{h \in H}} \rangle\!\rangle, \langle\!\langle \overleftrightarrow{[\![Comp_k]\!]^{k \in K}} \rangle\!\rangle, SV \rangle\!\rangle\end{array}}$$

With the auxiliary rule:

$$\frac{\forall l \in L.\, \mathcal{DM}_l = [\![m_l]\!]_{delegate}}{\mathcal{DMS}(m_l^{l \in L}) = \langle\!\langle \overleftrightarrow{\mathcal{DM}_l^{l \in L}} \rangle\!\rangle}$$

This rule defines the index sets that will be used in the vector Tables 1 and 3: H is the set of external and internal client interfaces, L is the set of all server and client methods, and K is the set of sub-components.

4.2.3. Future Proxies

The behaviour of future proxies for a composite component is slightly different from the one of primitive ones, as illustrated in Figure 8: the process *CProxy_m* in the figure gives the new value of the proxy semantics $[\![m]\!]_{proxy}$. The delegation methods create those proxies to remember the identifier of the future that the delegation method should serve. Consequently, the future proxy receives a future identifier and will return it as necessary upon request. The future proxy thus first receives a *New* action with a future identifier as parameter and then emits an *R_m(f)*. Such a proxy is somehow automatically recycled as, by construction, we know it is only used once.

4.2.4. Delegation Methods

The *Deleg_m* process, also shown in Figure 8 expresses the generation of delegate methods: $[\![m]\!]_{delegate}$ is given by the pLTS *Deleg_m*. This delegation process receives a *Call* invocation from the body, creates a future proxy, launches

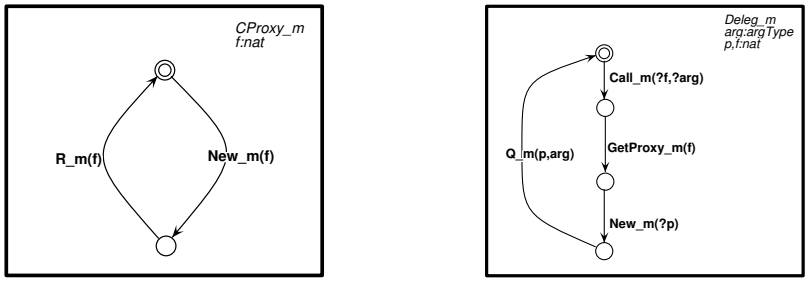

Figure 8: Auxiliary processes proxy and delegate of composite components

a remote invocation (either to an inner or to an external component) and finishes its execution. This way the composite component can continue its execution and serve another request, but the process of the future proxy is still running in order to redirect the reply towards the right future identifier. The *proxyManager* for composite component is not shown, indeed it is a direct adaptation of the primitive one (Figure 6): it behaves exactly the same except that the *GetProxy* action receives a future identifier as parameter, this parameter is then passed as argument in the *New* emission action (it will be used by the future proxy).

4.2.5. Synchronisation Vectors

Synchronisation vectors express first the internal wiring of the membrane of the composite component: they express the delegation mechanism, the coordination with the future proxy, the service of the requests, the creation of the future proxies, and the use of a future proxy to rename a future when it traverses the membrane of the composite. The sequence of actions is ensured by the behaviour of body and delegation methods while the synchronisation vectors ensure the coordination of all entities. The second kind of synchronisation vectors handle the routing of requests in the composite component; they express how requests follow the bindings and provide the semantics of the component composition.

Synchronisation vectors for composite components are organised into three sets: server-side (SV_S), client-side (SV_C), and binding-related (SV_B) synchronisation vectors. The set of server and client synchronisation vectors is the smallest set verifying the rules given in Table 2; they correspond to the synchronisation inside the membrane.

The server-side synchronisation vectors are defined by rules [C1] and [C2]. [C1] allows external components to enqueue a request in the queue, for each method of a server interface of the composite (given as first argument of SV_S). Note that replies (R_m) are not part of this rule because they depend on the bindings of the component; consequently they are treated among the binding synchronisation vectors. Rule [C2] uses a bigger set of methods as it takes into account requests of the server interfaces and the client interfaces of the composite; indeed, remember client interfaces have an associated internal server interface accessible by the sub-components of the composite. This second rule

Table 2: Server and client-side synchronisation vectors. The synchronised sub-pNets are (see Section 4.2.2): $\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{m \in \overline{m} \qquad f \in \mathbb{N}}{\langle iQ\_m(f, arg), -, -, -, - \rangle \rightarrow iQ\_m(f, arg) \in SV_S(\overline{m}, \overline{m'})} \quad \text{C1}$$

$$\frac{(i \in L_S \wedge m = m_i) \vee (i \in L_C \wedge m = m'_i) \qquad f \in \mathbb{N}}{\begin{array}{l}\{\langle Serve\_m(f, arg), Serve\_m(f, arg), -, -, - \rangle \rightarrow \underline{Serve\_m}(f, arg), \quad [1] \\ \langle -, Call\_m(f, arg), i \mapsto Call\_m(f, arg), -, - \rangle \rightarrow \underline{Call\_m}(f, arg)\ \} \quad [2] \\ \subseteq SV_S(m_l^{l \in L_S}, m_l'^{l \in L_C})\end{array}} \quad \text{C2}$$

$$\frac{j \in J \qquad m_k^{k \in K} = MethLabel(CItf_j) \qquad k \in K \qquad f, p \in \mathbb{N}}{\langle -, R\_m_k(f), k \mapsto Q\_m_k(p, arg), -, -, - \rangle \rightarrow Q\_m_k(p, arg) \in SV_C(CItf_j^{j \in J}, Itf_h^{h \in H})} \quad \text{C3}$$

$$\frac{\begin{array}{c}h \in H \qquad m_k^{k \in K} = MethLabel(Itf_h) \qquad k \in K \qquad f, p \in \mathbb{N} \\ \{\langle -, -, k \mapsto GetProxy\_m_k(f), h \mapsto k \mapsto GetProxy\_m_k(f), -, - \rangle \rightarrow \underline{GetProxy\_m_k}(f), \quad [1] \\ \langle -, -, k \mapsto New\_m_k(p), h \mapsto k \mapsto New\_m_k(p, f), h \mapsto k \mapsto p \mapsto New\_m_k(f), - \rangle \rightarrow \\ \underline{New\_m_k}(p, f)\} \quad [2] \\ \subseteq SV_C(CItf_j^{j \in J}, Itf_h^{h \in H})\end{array}}{} \quad \text{C4}$$

uses both arguments of SV_S, i.e. the list of client and server interfaces of the component. It deals with request service [C2.1], and subsequent calls [C2.2] to delegation pNets.

Client-side synchronisation vectors are expressed by rules [C3] and [C4] of Table 2. Similarly to the server case, [C3] is specific to external client interfaces (given as first argument of SV_C), while [C4] is applicable to both external and internal client interfaces (the second argument of SV_C). Remember the internal client interfaces are the symmetric of server interfaces of the composite component. The first rule exports request sending (Q_m) sent by delegate methods to the external components. Note that delegate methods are indexed by the method labels of the interfaces: in the *pNet* definition, $m_l^{l \in L}$ is a disjoint union, and thus each $l \in L$ is considered as equal to a method index k of a single interface i. Consequently, the request sending action (Q_m_k) is always issued by the delegate method indexed by k. Rule [C4] allows delegation methods to instantiate new proxies (by calls to the proxy manager and to the future proxies). Compared to the case of the primitive component, note the additional argument f passed to the proxy manager. This future identifier allows the future proxy (indexed p) to remember that the reply it will receive should be forwarded to the caller as the value for the future identifier f (and not p). In other words, the proxy remembers that the future p it will receive is in fact an alias for the future f. Similarly to primitive components, there are two actions for dealing with future proxy creations: *GetProxy* in [C4.1], and *New* in [C4.2].

Finally, the synchronisation vectors for the bindings of the composite com-

Table 3: Binding synchronisation vectors. The synchronised sub-pNets are (see definition in Section 4.2.2): $\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{\begin{array}{c}(CName.SI, C.SI_2) \in \overline{Binding} \qquad k \in K \qquad C = \text{Name}(Comp_k) \\ SItf_i = \text{GetItf}(CName.SI, CName < SItf_i^{i\in I}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding} >) \qquad i \in I \\ m_n^{n\in N} = \text{MethLabel}(SItf_i) \qquad n \in N \qquad m'_n = m_n\{\!\{SI \leftarrow SI_2\}\!\} \qquad q,\ f \in \mathbb{N}\end{array}}{\begin{array}{ll}\{\langle -, R\_m_n(f), n \mapsto Q\_m_n(q, arg), -, -, k \mapsto iQ\_m'_n(q, arg)\rangle \rightarrow \overline{Q\_m_n}(q, arg), & [1] \\ \ \langle -, -, -, i \mapsto n \mapsto Recycle\_m_n(q), i \mapsto n \mapsto q \mapsto R\_m_n(f), k \mapsto R\_m'_n(q, val)\rangle \rightarrow R\_m_n(f, val)\} & [2] \\ \qquad \subseteq SV_B(CName < SItf_i^{i\in I}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding} >)\end{array}} \text{ C5}$$

$$\frac{\begin{array}{c}(C.CI, CName.CI_2) \in \overline{Binding} \qquad k \in K \qquad C = \text{Name}(Comp_k) \\ CItf_j = \text{GetItf}(CName.CI_2, CName < \overline{SItf}, CItf_j^{j\in J}, Comp_k^{k\in K}, \overline{Binding} >) \qquad j \in J \\ m_n^{n\in N} = \text{MethLabel}(CItf_j) \qquad n \in N \qquad m'_n = m_n\{\!\{CI_2 \leftarrow CI\}\!\} \qquad p,\ f \in \mathbb{N}\end{array}}{\begin{array}{ll}\{\langle iQ\_m_n(f, arg), -, -, -, -, k \mapsto Q\_m'_n(f, arg)\rangle \rightarrow \overline{Q\_m_n}(f, arg), & [1] \\ \ \langle -, -, -, j \mapsto n \mapsto Recycle\_m_n(p), j \mapsto n \mapsto p \mapsto R\_m_n(f), k \mapsto iR\_m'_n(f, val)\rangle \rightarrow iR\_m_n(p, val)\} & [2] \\ \qquad \subseteq SV_B(CName < \overline{SItf}, CItf_j^{j\in J}, Comp_k^{k\in K}, \overline{Binding} >)\end{array}} \text{ C6}$$

$$\frac{\begin{array}{c}(C.CI, C'.SI) \in \overline{Binding} \qquad k, k' \in K \qquad C = \text{Name}(Comp_k) \qquad C' = \text{Name}(Comp_{k'}) \\ CItf\,' = \text{GetItf}(C.CI, CName < \overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding} >) \\ m_n^{n\in N} = \text{MethLabel}(CI) \qquad n \in N \qquad m'_n = m_n\{\!\{CI \leftarrow SI\}\!\} \qquad f \in \mathbb{N}\end{array}}{\begin{array}{ll}\{\langle -, -, -, -, -, (k \mapsto Q\_m_n(f, arg), k' \mapsto iQ\_m'_n(f, arg))\rangle \rightarrow \overline{Q\_m_n}(f, arg), & [1] \\ \ \langle -, -, -, -, -, (k \mapsto iR\_m_n(f, val), k' \mapsto R\_m'_n(f, val))\rangle \rightarrow \overline{R\_m_n}(f, val)\ \} & [2] \\ \qquad \subseteq SV_B(CName < \overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding} >)\end{array}} \text{ C7}$$

ponent are shown in Table 3. There are three rules for building SV_B. Rule [C5] deals with import bindings, i.e. bindings from the composite component's internal client interfaces to inner components. Symmetrically, [C6] concerns export bindings, from inner components to the composite component's internal server interfaces. The last rule [C7] specifies synchronisations due to bindings between two inner components.

The rule [C5] concerns import bindings. The first premise of the rule picks an import binding, the next premises find the concerned server interface of the composite component and the destination of the binding, i.e. a sub-component. The only remaining non-trivial premise is $m'_n = m_n\{\!\{SI \leftarrow SI_2\}\!\}$; it replaces in m_n the occurrence of the interface named SI by the interface SI_2. Indeed, remember that each MethodLabel contains the name of the invoked interface, this name must thus be updated when a request/reply/... is transmitted from an interface to another[10]. Similar premises, renaming an interface name, will also be used in rules [C6] and [C7]. The first item of Rule [C5.1] synchronises the emission of a request by a delegate method with the inner component bound to the concerned internal client interface. This action is also synchronised with the

[10]Other meta-information are encoded in the method label and should also be updated.

proxy for future that will receive the result computed by the request. The case [C5.2] concerns the corresponding reply that is issued by the inner component, this reply is sent to the outside of the composite component. Note the particular flow of information here: the inner component emits a value for future q, that is directly synchronised with the future proxy number q of the composite component; the identifier of the future to be sent to the outside becomes f; it is retrieved from the future proxy, and an action $R\_m'_n(f, v\ al)$ is emitted. At the same time, a recycling action is triggered in the proxy manager.

The second rule [C6] manages export bindings, it also has one item for request emission [C6.1] and another one for reply reception [C6.2]. A request emitted by the inner component on the first side of the binding is enqueued in the encompassing composite component (at the other side of the binding). Replies are redirected when received by the encompassing component: when the reply for future p is received, the future proxy at index p is used to retrieve the future identifier f, and finally the result *val* is transmitted, associated with the future f, to the inner component indexed by k. At the same time, a *Recycle* action is triggered in the adequate proxy manager.

Rule [C7] deals with bindings between two inner components. It considers a binding between an interface of component C and an interface of component C'. It finds k, the index of C, and k', the one of C'; the rule directly transfers requests [C7.1] and replies [C7.2] from one component to the other for all the methods of the client interface bound. Like in the preceding rules, the name of the interface is updated during transmission.

This section presented a behavioural semantics for hierarchical components communicating by asynchronous requests. The behaviour of composite components is only to forward requests to the adequate destination. We encode replies by means of futures, and composite components act as reply forwarders. This section generalised the examples of specifications one could find in [5]. In previous work, we provided a behavioural model for some given GCM applications [9, 7]; these models have been used to prove the correct behaviour of the analysed application. These examples show the capacities of our behavioural models, and their adequacy for verification purposes. This section generalises those previous examples and instances, in order to formally specify the automatic generation of behavioural models for asynchronous components. The next section will define first-class futures, allowing more asynchronous behaviour. First-class futures generalise the future forwarding mechanism that we presented above to allow futures to be passed as method arguments and return values between any two components.

5. First-class Futures

In this section we focus in particular on first-class futures, which are a crucial aspect of GCM components featuring wait-by-necessity mechanism. We will introduce behavioural models for *first-class futures* in order to allow for more asynchrony between components.

We will specify new rules for the extended component behaviour. In practice, we build new behavioural semantics by modifying the semantics defined in the preceding section, for this we use two additional operators that simplify the modification and composition of pNets. The first one allows us to remove some part of the synchronisation vectors in order to redefine them; the second one allows us to extend a pNet definition with new sub-pNets and/or new synchronisation vectors.

- $\oslash$: Let *pNet* be a pNet and $\mathcal{A}$ be an indexed set of labels; $pNet \oslash \mathcal{A}$ returns a pNet similar to *pNet* but with restricted synchronisation vectors. The synchronisation vectors of $pNet \oslash \mathcal{A}$ are the ones of *pNet* except all the synchronisation vectors containing an element of $\mathcal{A}$ as part of their global synchronisation label. For example, if m belongs to $\mathcal{A}$ then all the vectors containing m in their global label will be removed, in that case the labels concerned would be: Q_m, iQ_m, $\underline{Q\_m}$, R_m, $Serve\_m$, etc. Remember that method labels contain the name of the interface that contains the method and consequently, removing a method label cannot remove an action concerning another method with the same name in another interface. This operator allows us to remove all the synchronisation vectors of a pNet that are related to one aspect of the synchronisation (e.g., the method invocation on some methods) in order to set a new synchronisation pattern (e.g. intercept this method invocation).

- $\oplus$: For $I \in \mathcal{I}$ and $I' \in \mathcal{I}$ disjoint, let $pNet = \langle\!\langle pNet_i^{i \in I}, SV_k^{k \in K} \rangle\!\rangle$ be a pNet and ${pNet'}_i^{i \in I'}$ be a pNet family (possibly empty). Let ${SV'}_k^{k \in K'}$ be a set of synchronisation vectors over $I \uplus I'$, such that for $k \in K'$, $SV'_k = \alpha_j^{j \in J_k} \rightarrow \alpha'_k$ where $\alpha'_k \in \mathrm{Sort}(pNet)$, $J_k \in \mathcal{I}$, $J_k \subseteq I \uplus I'$, and $\forall j \in J_k \cap I.\, \alpha_j \in \mathrm{Sort}(pNet_j)$, and $\forall j \in J_k \cap I'.\, \alpha_j \in \mathrm{Sort}(pNet'_j)$. $pNet \oplus \langle\!\langle {pNet'}_i^{i \in I'}, {SV'}_k^{k \in K'} \rangle\!\rangle$ extends *pNet* with the new sub-pNets $pNet'_i$. The synchronisation vectors are kept (they do not synchronise the new sub-pNets); and the new synchronisation vectors ${SV'}_k^{k \in K'}$ are added to the ones of *pNet*:

$$pNet \oplus \langle\!\langle {pNet'}_i^{i \in I'}, {SV'}_k^{k \in K'} \rangle\!\rangle = \langle\!\langle pNet_i^{i \in I} \uplus {pNet'}_i^{i \in I'}, SV_k^{k \in K} \uplus {SV'}_k^{k \in K'} \rangle\!\rangle$$

5.1. Informal Semantics of First-class Futures

We call *first-class futures*, the futures that can be transmitted between components before their value is known. Without first-class futures, a component must wait for the result of a request and perform a *GetValue* before being able to send this result to another component (in a request parameter, or in a request result). With first-class futures, a component can send a *generalised reference* to the future, i.e. a reference that uniquely represents the future in the whole component system. This way, once the result is computed it is sent to all the components that hold a (generalised) reference to the corresponding future.

In the model presented in Section 4, composite components act as reply forwarders which correspond to some kind of limited first-class futures. However,

there are two differences concerning our use-cases. First, we need a somehow simpler mechanism to transmit a future value *without renaming a future*, only based on its global reference. Second, we also need the primitive components to be able to handle first-class futures. This means primitive components must be able to know futures and have future proxies for futures that were not created by them (they were received as method parameters); consequently new proxies have to be created for first-class futures with a slightly different mechanism for creating and fulfilling those futures. Finally, a forwarding mechanism similar to the one defined in the semantics of ASP (forward-based future update strategy in [32]) has to be added to transmit the future value not only to the component that performed the method invocation but also to the other components that have received the generalized reference to the future.

We will consider in this section only the case when a future is transmitted as the single argument of a request. The case where a single future is returned as request result is similar, and closer to the reply forwarding mechanism implemented by the composite components. The case where a future is only a part of the transmitted object requires reasoning on the abstract representation of the transmitted objects, which we do not do in this article. Overall, we consider here the minimal case which is sufficient and general enough to understand the mechanism of the generation of behavioural models for first-class futures. A general study on the different kind of first-class futures, their identification and a few usage scenarios more complex than the one presented here can be found in [8]; however, compared to [8], the approach presented here provides a complete formal specification of the generation of behavioural models whereas the previous study illustrated the kind of properties we are able to prove with our model for first-class futures. We suppose in this section that the methods that can receive first-class futures have been previously identified.

To handle first-class futures, it is not possible anymore to consider future identifiers as integers that are locally unique. Future identifiers need to be global references where uniqueness is guaranteed globally in the system. For this, we defined generalised references, which identify uniquely future proxies. A generalised reference *gf* is defined by a tuple: component name, interface, method, future identifier. Generalised references belong to the set of valid request arguments (denoted by the variable *arg*), and can be used as a reference to the future that can be transmitted to other components.

5.2. Principles and Illustrative example

Figure 9 is a simple scenario demonstrating the transmission of a future value as a parameter of a remote request. The principle of this scenario is the following. The component A first invokes a request on the component B (step 1); then the result of this invocation is sent as the single parameter of a request invocation to the component C (step 2). Possibly, the component C can try to access the future received, resulting in a wait-by-necessity (step 3). When the result is computed by B, it is returned to A and then to C (step 4), which releases the wait-by-necessity.

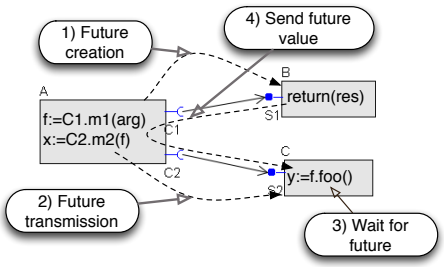

Figure 9: Scenario of first-class future transmission

The pNet for the scenario described above is shown in Figure 10. The corresponding rules will be detailed in Section 5.3 and Tables 4 and 5. The figure focuses on aspects related to the transmission of the future (denoted f in the code snippet, and indexed p as a local proxy) and its update. Let gf be the corresponding generalised reference. The figure shows how a generalised future reference can be sent between two components, and handled afterwards. Compared to the usual future proxy mechanism in Figures 4 and 7, it shows new proxies for futures that can be sent and received as request parameters, as detailed below:

Considering the emitting component, the proxy *FProxy_m1[p]* for the local future is slightly different; the only addition is that it is able to emit a *Forward*(gf, val) action, when this is requested by the remote components C.

The generalised reference gf is sent upon the invocation of *Q_m2* (see the arrow from *Method_m* in A to *Queue* in C). It is built by the function call *GeneralisedRef*($CName$, Itf, $MethodLabel$, p).

On the receiver side, when the request is handled by the component C, the call to the local method is intercepted by a specific pNet *FutDetect_m**. This pNet creates a local proxy for representing gf (*Proxy_F1*, indexed locally by q); q then replaces gf in the invocation. This local future proxy waits for the *iForward* communication coming from the remote proxy, and can then be accessed by the service method *Method_m2*. We have a single *PM_F1* proxy manager for all the first-class proxies (*Proxy_F1*), it acts as any other proxy manager except that the first-class proxy is given the generalised reference gf when defining the semantics, and it transmits gf when creating the proxy. It thus has a similar behaviour to the proxy manager for a composite component; it does not correspond to any method name, we re-use the proxy manager semantics, giving it a pseudo method name: *F1*. Consequently $PM\_F1 = [\![F1]\!]_{proxyManager}$.

In primitive components, two changes have to be made. First each primitive component that can receive a first-class future is equipped with first-class future proxies and a first-class proxy manager. A new kind of proxy, *Proxy_F1*, needs to be created; those proxies act as a local future proxy for a proxy received as request parameter. The first-class future proxy receives the future value thanks to an *iForward* event. Second, at the other side, proxies of futures that

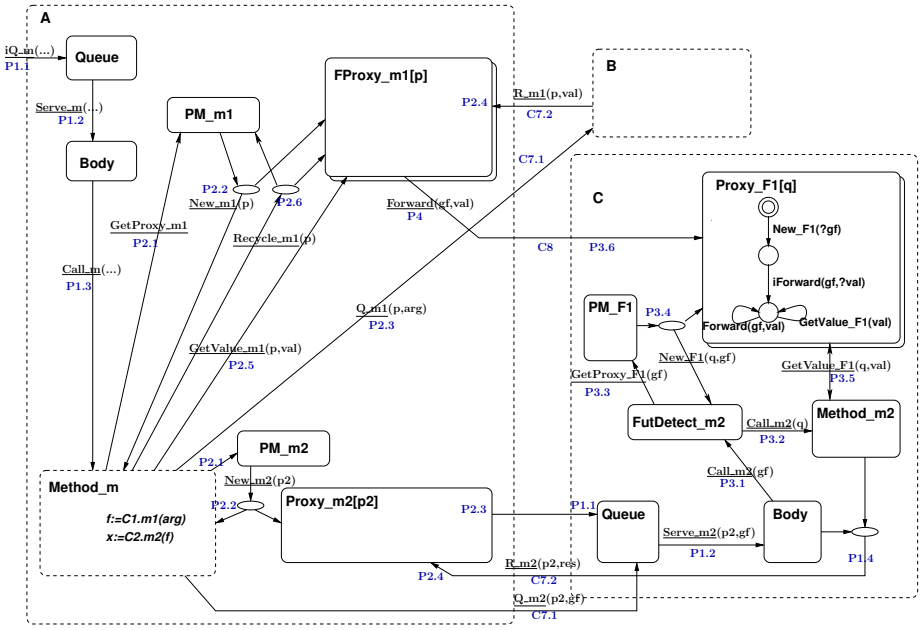

Figure 10: Transmitting a future as a method call parameter

can be transmitted are unchanged but the synchronisation vector of primitive components must be able to forward the value of a future if this future can be forwarded. We create a *Forward* action label for that. Note that any component that received a first-class future can send it again, this is why first-class future proxies must also be able to send a *Forward* event.

Concerning composite components, they do not need specific future proxies as they never need the value of a future. Instead they must transmit adequately the *Forward* communication from the component that holds the proxy created at method invocation to the component that may receive the first-class future, triggering the *iForward* action in the first-class proxy.

In the GCM/ProActive implementation a similar mechanism occurs at runtime. Indeed when a future is received as a request parameter, during the deserialisation of the future reference, a local proxy for future is created, awaiting for the reception of the future value.

5.3. Primitive Components: pNets and Synchronisation Vectors

This section defines $[\![\ ]\!]^{F1}$ the semantics of components with first-class futures. As shown in the illustrative example, the pNet of the primitive component must be extended with proxies for storing locally first-class futures received as parameters, called *Proxy_F1* in Figure 10 and in the rules below, and a proxy manager managing those proxies. The proxy manager for first-class futures is very similar to the ones shown before; like for composite components it receives

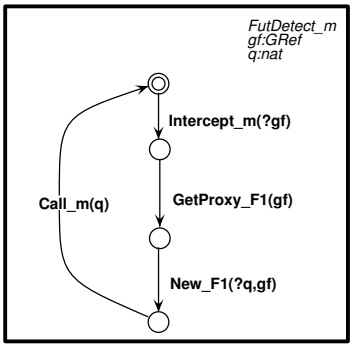

Figure 11: Proxy for a first-class future and Future Detector machine

a future reference and transmits it upon proxy creation, except this time the reference is a generalised one. Let *GRef* be the set of all generalised references and let *gf* range over *GRef*. *GeneralisedRef* is a constructor of generalised references:

$$GeneralisedRef : CName \times Itf \times MethodLabels \times \mathbb{N} \to GRef$$

Figure 11 shows the pLTS of a *FutDetect* process that intercepts the local service of requests with a future as parameter. It creates a local proxy before delegating the call. *FutDetect_m* defines the pLTS denoted by $[\![m]\!]_{FutDetect}$ in the following. The handling of a service method that can receive a future as a parameter is managed by an intermediate *FutDetect_m* process. This process, denoted FD_l below, creates a local proxy representing the transmitted future.

$$\frac{\begin{array}{c} m_l^{l\in L} = \mathrm{MethLabel}(\overline{SItf}) \qquad \text{the methods } m_l^{l\in L'} \text{ can receive a future as parameter} \\ PM\_F1 = [\![F1]\!]_{proxyManager} \\ \forall l \in L'.\ FD_l = [\![m_l]\!]_{FutDetect} \qquad SV^{F1} = SV_S^{F1}(m_l^{l\in L'}, L) \cup SV_C^{F1}(\overline{CItf}, CName) \end{array}}{\begin{array}{l} [\![CName < \overline{SItf}, \overline{CItf}, \overline{M} >]\!]^{F1} = [\![CName < \overline{SItf}, \overline{CItf}, \overline{M} >]\!] \oslash call\_m_l^{l\in L'} \\ \qquad \oplus \quad \langle\!\langle \overleftrightarrow{FD_l{}^{l\in L'}} \rangle\!\rangle, PM\_F1, \langle\!\langle \overleftrightarrow{Proxy\_F1_n{}^{n\in\mathbb{N}}} \rangle\!\rangle, SV^{F1} \rangle\!\rangle \end{array}}$$

The new pNet extends the old one thanks to the operator $\oplus$ defined previously. Concerning methods that can receive a future as parameter, direct *Call* invocations from the body to the service methods are removed by the $\oslash$ operator. The new synchronisation vectors are defined in Table 4.

There are seven entries in the new synchronisation vectors of Rule [P3]. The two first ones, [P3.1] and [P3.2] intercept the invocation from the body to the service method, those invocations now are intercepted by the *FutDetect* process. Note that what was previously a "call" action from the body to the service method is now split into two steps. We use an *Intercept* action and a *Call* action to distinguish the interaction between the *FutDetect* process and the body from the interaction between the *FutDetect* process and the service method. [P3.3] and [P3.4] deal with the creation of proxies for futures received as argument, which is quite similar to a classical proxy creation, shown in Section 4.1. All the

Table 4: Synchronisation vectors related to client and server interfaces of primitive components for first class futures. The synchronised sub-pNets are defined above:
$\langle\!\langle Queue, Body, ServiceMethods, ProxyManagers, Proxies, FutDetect, PM\_F1, Proxies\_F1 \rangle\!\rangle$

$$\frac{i \in L' \qquad gf \in GRef \qquad q \in \mathbb{N} \qquad j \in L}{\begin{array}{l}\{\langle -, Call\_m_i(gf), -, -, -, i{\mapsto}Intercept\_m_i(gf), -, - \rangle \rightarrow \underline{Call\_m_i}(gf), \quad [1]\\ \langle -, -, i{\mapsto}Call\_m_i(q), -, -, i{\mapsto}Call\_m_i(q), -, - \rangle \rightarrow \underline{Call\_m_i}(q), \quad [2]\\ \langle -, -, -, -, -, i{\mapsto}GetProxy\_F1(gf), GetProxy\_F1(gf), - \rangle \rightarrow \underline{GetProxy\_F1}(gf), \quad [3]\\ \langle -, -, -, -, -, i{\mapsto}New\_F1(q, gf), New\_F1(q), q{\mapsto}New\_F1(gf) \rangle \rightarrow \underline{New\_F1}(q, gf), \quad [4]\\ \langle -, -, j{\mapsto}GetValue\_F1(q, val), -, -, -, -, q{\mapsto}GetValue\_F1(val) \rangle \rightarrow \underline{GetValue\_F1}(q, val), \quad [5]\\ \langle -, -, -, -, -, -, -, q{\mapsto}iForward(gf, val) \rangle \rightarrow iForward(gf, val), \quad [6]\\ \langle -, -, -, -, -, -, -, q{\mapsto}Forward(gf, val) \rangle \rightarrow Forward(gf, val)\} \quad [7]\\ \subseteq SV_S^{F1}(m_i^{i \in L'}, L)\end{array}} \text{ P3}$$

$$\frac{\begin{array}{c}j \in J \qquad m_i^{i \in I} = \text{MethLabel}(CItf_j) \qquad i \in I\\ \text{Futures for requests on } m_i \text{ can be sent as parameter}\\ p \in \mathbb{N} \qquad gf = GeneralizedRef(CName, CItf_j, m_i, p)\end{array}}{\langle -, -, -, -, j{\mapsto}i{\mapsto}p{\mapsto}GetValue\_m_i(val), -, -, - \rangle \rightarrow Forward(gf, val) \in SV_C^{F1}(CItf_j^{j \in J}, CName)} \text{ P4}$$

service methods can access the *Proxy_F1* proxies by a *GetValue_F1*[11], which is expressed by [P3.5]. The new future proxies are updated by a *iForward* action [P3.6], instead of a *iR* action for usual proxies. Finally, [P3.7] emits a forward from the first-class future proxy. Indeed, if the component that received a first-class future itself forwards the future to another component, it also needs to forward the future value when its value is received.

Rule [P4] defines new synchronisation vectors on the client side of the primitive component; they allow the emission of *Forward* actions. When a future proxy of kind *FProxy_m* emits a *GetValue*, i.e. as soon as the future value is known, a *Forward* event can be emitted by the component. The proxy defined in Section 4.1.5 is unchanged, only additional synchronisation vectors are added for first-class futures: when building the synchronisation vector of the primitive component, a generalised reference is computed from the name of the corresponding interface and method, it is used to send a *Forward* event to the outside of the component.

5.4. Composite Components: New Synchronisation Vectors

Concerning composite components, the composition of the pNets is unchanged, we only add new synchronisation vectors for transmitting *Forward*

[11] If the component is stateful then the future references can be transferred between two service methods; this is why we consider that the generalised reference can be accessed from any service method.

Table 5: Binding synchronisation vectors for first class futures. The synchronised sub-pNets are unchanged from the behavioural semantics of composite components (see Section 4.2.2): $\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{\begin{array}{c}(C.CI, C'.SI) \in \overline{Binding} \\ C \neq CName \qquad k, k' \in K \qquad C = \text{Name}(Comp_k) \qquad C' = \text{Name}(Comp_{k'}) \\ CItf' = \text{GetItf}(C.CI, CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >) \\ gf \in GRef \qquad \exists m_n \in \text{MethLabel}(CItf').\ m_n \text{ can pass a future as parameter}\end{array}}{\begin{array}{r}\langle -,-,-,-,-, (k \mapsto Forward(gf, val), k' \mapsto iForward(gf, val)) \rangle \rightarrow \underline{Forward}(gf, val) \\ \in SV_B^{F1}(CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >)\end{array}} \text{C8}$$

$$\frac{\begin{array}{c}(CName.SI, C.SI_2) \in \overline{Binding} \qquad k \in K \qquad C = \text{Name}(Comp_k) \\ SItf' = \text{GetItf}(CName.SI, CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >) \\ gf \in GRef \qquad \exists m_n \in \text{MethLabel}(SItf').\ m_n \text{ can pass a future as parameter}\end{array}}{\begin{array}{r}\langle -,-,-,-,-, k \mapsto iForward(gf, val) \rangle \rightarrow iForward(gf, val) \\ \in SV_B^{F1}(CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >)\end{array}} \text{C9}$$

$$\frac{\begin{array}{c}(C.CI, CName.CI_2) \in \overline{Binding} \qquad C = \text{Name}(Comp_k) \\ CItf' = \text{GetItf}(C.CI, CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >) \\ gf \in GRef \qquad \exists m_n \in \text{MethLabel}(CItf').\ m_n \text{ can pass a future as parameter}\end{array}}{\begin{array}{r}\langle -,-,-,-,-, k \mapsto Forward(gf, val) \rangle \rightarrow Forward(gf, val) \\ \in SV_B^{F1}(CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >)\end{array}} \text{C10}$$

actions between components:

$$[\![CName < \overline{SItf}, \overline{CItf}, \overline{Comp}, \overline{Binding} >]\!]^{F1} = [\![CName < \overline{SItf}, \overline{CItf}, \overline{Comp}, \overline{Binding} >]\!] \oplus \langle\!\langle SV_B^{F1}(\overline{Binding}, \overline{SItf}, \overline{CItf}, \overline{Comp}) \rangle\!\rangle$$

Table 5 defines the transmission of forward events. Rule [C8] deals with brother bindings: consider two components at the same level, if one forwards a future value, the other should receive it. Rule [C9] (resp. [C10]) transmits forward reception (resp. emission) along import (resp. export) bindings.

Note that generating synchronisation for all possible generalised reference will not scale in practice but each $gf \in GRef$ can easily be restricted; indeed the origin of the future (component, method label) is known and the future flow can be approximated.

6. Validation

The purpose of this section is to provide arguments validating the semantics given in this article. The first part shows formally that we did not forget any synchronisation in our formalisation process; the second part focuses on an example to show how our behavioural specification works in practice.

6.1. Correctness Theorems

The objective of this section is to exhibit two theorems that can be used to assert the validity of our semantics. Proving the equivalence between our behavioural semantics and another semantics for GCM components (like [13]) is out of the scope of this article, but we will briefly sketch the points of comparison between the two semantics.

We first prove that our semantics features a complete synchronisation between the pNets. More precisely, we show that all the actions emitted or received by a pNet are synchronised and composed with pNets able to emit the corresponding actions. This can only be verified if all the interfaces of the component are connected so that every message that can be received (resp. emitted) has a source (resp. a target). Hence, we will reason on *fully connected components*. Additionally, due to sub-typing some methods can belong to a server interface and not be invocable by the interface bound to this server interface. To simplify the formulation of the theorem, we restrict ourselves to the case where no sub-typing is allowed, i.e. the sub-typing relation $\trianglelefteq$ is the identity.

Fractal and GCM components support the notion of mandatory and optional interfaces. A component cannot be started if one of its mandatory interfaces is not connected. Based on this notion, we define a fully connected component as a component that has all its interfaces bound, it corresponds to a Fractal component that only has mandatory interfaces.

Definition 3 (Fully connected component). We state that a composite component is *fully connected* (denoted by $FC(C) = true$) if all its inner interfaces and all the interfaces of the components it contains are connected, and also all the components it contains are themselves (recursively) fully connected. Primitive components are always considered fully connected. Formally:

The component C is fully connected if either C is a primitive, or C is a composite of the form $C = CName < \overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding} >$ and

$$FC(C) \quad \Leftrightarrow \quad \begin{cases} WF(C) \quad \wedge \quad \forall k \in K.\, FC(Comp_k) \wedge \\ \forall SItf \in \overline{SItf}.\, \exists QN.\, (CName.SItf, QN) \in \overline{Binding} \wedge \\ \forall CItf \in \overline{CItf}.\, \exists QN.\, (QN, CName.CItf) \in \overline{Binding} \wedge \\ \forall k \in K. \forall Itf \in \text{Interfaces}(Comp_k). \\ \quad \exists QN.\, (Comp_k.Itf, QN) \in \overline{Binding} \vee (QN, Comp_k.Itf) \in \overline{Binding} \end{cases}$$

We now define the set of unsynchronised action terms, in order to prove that this set is empty (or reduced to an acceptable set of actions).

Definition 4 (Unsynchronised actions). An action is unsynchronised in pNet P if it is either 1) in the sort of a sub-pNet of P but it is never used in a synchronisation vector[12], or 2) in the sort of P but the synchronisation

[12]To formalise the case 1 we use *valuations* (denoted ϕ) which are mappings from variable names to expressions. They are used here to check that there is an instantiation of the variables such that an action of a sub-pNet matches a label of the synchronisation vector.

vector producing this action can never be triggered, or 3) an unsynchronised action of one sub-pNet of P.

$$
\begin{aligned}
&\mathit{Unsynchronised}(\langle\!\langle pNet_i^{i\in I}, \overline{SV}\rangle\!\rangle) = \\
&\quad \bigcup_{i\in I} \{a \in \mathrm{Sort}(pNet_i) | \forall \alpha_j^{j\in J_k} \rightarrow \alpha'_k \in \overline{SV}. \\
&\qquad\qquad (i \notin J_k \vee \nexists \phi, \phi'. \alpha_i\phi = a\phi' \vee \exists j \in J_k. \nexists \phi, \phi'. \alpha_j\phi \in \mathrm{Sort}(pNet_j)\phi')\} \\
&\quad \cup \{a \in \mathrm{Sort}(\langle\!\langle pNet_i^{i\in I}, \overline{SV}\rangle\!\rangle) | \forall \alpha_j^{j\in J_k} \rightarrow \alpha'_k \in \overline{SV}. \\
&\qquad\qquad (\alpha'_k \neq a \vee \exists j \in J_k. \nexists \phi, \phi'. \alpha_j\phi \in \mathrm{Sort}(pNet_j)\phi')\} \\
&\quad \cup \bigcup_{i\in I} \mathit{Unsynchronised}(pNet_i)
\end{aligned}
$$

A pLTS is always fully synchronised because all actions of the sort can be emitted (see Definition 2):

$$\mathit{Unsynchronised}(\langle\!\langle S, s_0, L, \rightarrow\rangle\!\rangle) = \emptyset$$

A queue is always fully synchronised:

$$\mathit{Unsynchronised}(\mathit{Queue}(\overline{m})) = \emptyset$$

Note that this definition is quite static, because an action is in the sort of a pLTS if it is the label of one of its transitions, and an action is in the sort of a pNet if it is produced by one synchronisation vector. Additionally, *Unsynchronised* does not deal with the actions that are indirectly unsynchronised. Indeed if an action is unsynchronised then all the synchronisation vectors triggered by this action will never be triggered, however the resulting action of those un-triggered synchronisation vectors will not belong to $\mathit{Unsynchronised}(\llbracket C \rrbracket)$ because the unsynchronised actions of a pNet belong to its sort.

A basic correctness theorem for our semantics is that, if a component is fully connected, then there is no unsynchronised action in the behavioural semantics except the ones involving subtyped method labels or the ones involving methods that are never used. The proof of this theorem is presented in Appendix C.

Theorem 1 (Complete synchronisation) *If the component C is fully connected, i.e. if $FC(C) = true$, then*

$$\mathit{Unsynchronised}(\llbracket C \rrbracket) \subseteq \{Q\_m_i \,|\, m_i \text{ is never invoked}\}$$

In particular, if all methods of client interfaces of primitive components are called by at least one service method, then $\mathit{Unsynchronised}(\llbracket C \rrbracket) = \emptyset$.

The definitions presented in this section are static. In particular, we did not check that actions in the LTS are reachable. Concerning the pLTSs we define in this article, this can easily be checked. Concerning (user-defined) service methods, verifying reachability of actions is not decidable. However we suppose here that method bodies start by *Call* and end by *R*, and that remote invocation are preceded by a proxy creation (Request emission is encoded by the sequence: *GetProxy*, *New*, *Q*). These well-formedness conditions can automatically be checked.

Discussion about the theorem. The applicability conditions of Theorem 1 are quite restrictive. To be closer to its practical application, the theorem should be strengthened in two ways. 1) *Sub-typing*: In our tools, sub-typing between interfaces is allowed, generating some input actions that are never triggered. 2) *Unbound server interface*: It is safe to have not fully connected components and allow server interfaces not to be bound (provided all client interfaces are). In both cases, the theorem can be generalised and written such that some of the actions for incoming requests are never triggered. Such a generalisation of the theorem raises no technical difficulty concerning the proof but makes the theorem statement more complex. In the generalised theorem, the set of unsynchronised actions would also include actions related to the methods that belong to the interface of a sub-type but not to the corresponding super-type, and actions related to unbound server interfaces.

Theorem 1 ensures that, in the model, we generate all the necessary synchronisation vectors. This is the first step to ensure that bugs that are found when model-checking the generated pNet model are due to errors in the specification of service methods or in the design of the distributed system, rather than bugs in our generation rules. The theorem above is however not sufficient to ensure this property: we additionally need to prove that the necessary actions can be triggered by the different pNets involved in the composition. The theorem below will prove this property concerning the routing of requests. The other operations (replies, service of requests, etc.) should be proven correct in the same way.

Correct routing of Requests

We will prove that whenever the pNet representing a component sends a request on one of its client interfaces, then the pNet encoding its environment will indeed transmit this request properly (through all required queues, proxies, and component boundaries) to a pNet representing a primitive component able to receive and process this request.

We start with a lemma stating that each emitted request will be received by another component. This shows that requests can follow one binding. In a second step we will prove that the request arrives at a primitive component (where it will be treated) by recursively applying the first lemma. To transmit a request to its destination, several bindings and intermediate components have to be traversed. In order to ensure the recursive applicability of the first lemma, we also prove that every composite that receives a request will emit a new one through its *Deleg* pNet.

This section relies on the semantics of pNets defined in Appendix A. The semantics of a pNet $\|pNet\|_\Phi$ is an LTS where states are tuple representing the states of the pLTSs composing *pNet*. In this section, we use the notation $\ddot{s}_{pNet}$ to refer to the set of reachable states of the semantics of *pNet*. $\rightarrow$ denotes the transition of the semantics $\|pNet\|_\Phi$. Also, we consider that we have infinite families of future proxies so that it is always possible to create a new one. We use $s_i \subseteq s$ where s, s_i are pNet states to stand for s_i is a sub-tuple of s.

Let us consider a complete model of an application *pNet*, let us denote by $\overline{pNet}$ the set of pNets, representing only components (primitive and composite components) that are included in *pNet*. In fact, a pNet is formed of a tree of pNets: those representing components, and others added for handling the various mechanisms of the components (routing, forwarding, delegating...), but also the intermediate pNets that are built by the $\langle\!\langle \overleftrightarrow{PN_i^{i\in I}} \rangle\!\rangle$ operator to encapsulate pNet families. Additionally, for any $pNet \in \overline{pNet}$, when *pNet* is a composite component, we denote *Deleg*(*pNet*) the set of delegation sub-pNets of *pNet* and $\overline{Deleg(pNet)}$ the set of delegation sub-pNets of all pNets in $\overline{pNet}$.

Lemma 1 (Request routing)
Let $pNet_i \in \overline{pNet} \cup \overline{Deleg(pNet)}$. *Let* $s_i, s_i' \in \ddot{s}_{pNet_i}$ *such that* $s_i \xrightarrow{Q\_m} s_i'$ *then* $\exists pNet_j \in \overline{pNet}$, $\exists s_j, s_j' \in \ddot{s}_{pNet_j} s_j \xrightarrow{iQ\_m} s_j'$ *and* $\exists s, s' \in \ddot{s}_{pNet}.s \rightarrow^* \xrightarrow{Q\_m} s'$ *where* $s_i \subseteq s$, *and* $s_j \subseteq s$, *and* $s_i' \subseteq s'$, *and* $s_j' \subseteq s'$.

Additionally, if $pNet_j$ *is the pNet of a composite component then* $\exists s_j'' \in \ddot{s}_{pNet_j}.\, s_j' \rightarrow^* s_j''$ *such that* $\exists s_k, s_k' \in \ddot{s}_{Deleg(pNet_j)}.\, s_k \xrightarrow{Q\_m} s_k'$.

This lemma states that each request emitted by either a component or a delegation method will arrive at a destination component where the request reception is synchronized with the request emission. Each element involved in the communication progresses into a state of the global pNet that represents the instant when the communication has been performed.

Proof Sketch. We proceed in two steps, first we prove the part of the lemma preceding the "additionally".

We then distinguish the case where the request is emitted from the pNet of a component or from the pNet of a *Deleg*. We start by the case where the emitter $pNet_i$ is the pNet of a component C. The request must correspond by construction to the signature of a client interface. The binding is defined in the super-component of C in the hierarchy. Due to well-formedness the binding is ensured to exist and the destination interface to provide the method m. Two cases are possible depending on which interface is bound to the client interface[13].

• **Brother bindings:** In this case, the request Q_m is emitted from the pNet of a component that is connected to another component of the same hierarchical level following a brother binding of the form $(C.CI, C'.SI)$ where C and C' are two sub-components of the same composite. As m belongs to the interface SI, the pNet of the destination component C' must accept m in its queue and provide the action iQ_m, and rule [C7.1] encodes the synchronisation between the emitter and the receiver. This proves the main statement of the lemma

[13]This corresponds at the same time to the two possibilities offered by the component structure and to the synchronisation vectors able to synchronise an action Q_m.

where $pNet_j$ is the pNet of the component C'[14]. The global action $\underline{Q\_m}$ is emitted.

• **Export bindings:** If the request Q_m is emitted from the pNet of a sub-component to a parent composite: binding of the form $(C.CI, C'.SI)$ where C is a sub-component of C'. The request can be received in the queue of the super-component C' as iQ_m, and rule [C6.1] encodes the synchronisation between them, and returns to the top-level the action $\underline{Q\_m}$.

If the request Q_m is emitted from the pNet of a *Deleg* either to an inner or to an external component, two cases are again possible.

• *Deleg* **for a server interface:** This corresponds to a request following an **import binding**. The synchronisation vector defined by rule [C5.1] allows synchronisation with inner sub-components. The inner sub-component must provide an iQ_m action for the same reasons as above.

• *Deleg* **for a client interface:** The synchronisation vector defined by rule [C3] allows synchronisation with external components; more precisely, this synchronisation vector allows the pNet of the composite component containing the *Deleg* pNet to emit a Q_m action. Then we can apply the case where Q_m is emitted by the pNet of a component (detailed above).

Finally, we prove the "additionally" clause of the theorem. The objective is to show that, for any request entering the queue of a composite component, this request will be eventually emitted by a *Deleg* pNet. This part is proved by induction on the number of elements before the request Q_m in the queue: if Q_m is first, we will show that the request can be emitted by a *Deleg*, if it is not then the same arguments show that another request can be dequeued and thus by recurrence we show that eventually the request Q_m is served and emitted by a *Deleg* pNet.

Suppose that all the involved pLTSs (body, Deleg, not-yet allocated futures, proxy manager) are in their initial state. When the *Body* pLTS is in its initial state, it serves the oldest request of the queue (enabled by rule [C2.1]). Then, it synchronises with the *Deleg* pLTS (by synchronisation vector rule [C2.2]) that starts its execution and then requires a new future proxy (synchronised by rules [C4]); the creation of a future proxy necessarily succeeds because we supposed that the proxy families are infinite (and proxy manager is well-synchronized with the future proxy and the *Deleg* pNet). After those steps the *Deleg* pNet sends a Q_m action. Note that after this emission of Q_m action, the *Deleg*, *Body*, and proxy manager pLTSs return to their initial states.

The paragraph above supposed that the concerned pNets are in their initial state and proved that the delegation process succeeds. As the *Body* pLTS only serves requests that will be delegated, if the prerequisite for this process to succeed is not met and the*Body* pNet is not in its initial state then it is currently delegating a request; it is in the middle of the delegation process. The paragraph above ensures that it will eventually return to its initial state and be ready to

[14]Note that the queue is always able to receive the incoming request, whatever the state of $pNet_j$ is.

serve another request. The same reasoning applies to the *Deleg* pLTS and the proxy manager that also return to their initial state.

Lemma 2 (No infinite loops) *For a well-formed GCM system, and a method m, in the (finite) tree $\overline{pNet}$ of the pNets representing components, consider the transitions $\xrightarrow{Q\_m}$ occurring in Lemma 1. Then all $\xrightarrow{Q\_m}$ chains reach, in a finite number of steps, a state of the global pNet representing the application state where a primitive component has received the request Q_m.*

Proof. From the well-formedness definition, we know that bindings cannot relate a client and a server interface of the same component. This forbids immediate loops of $\xrightarrow{Q\_m}$, inside or outside the pNet representing a composite.

Then we have to prove that we have no infinite loops going only through composite components pNets. Observing the cases from the proof of Lemma 1:

- when a $\xrightarrow{Q\_m}$ transition is going up in the component hierarchy (export binding), it corresponds to a request emitted by a client interface of a component, so in the context of its parent composite, it can either go the a sibling sub-component (horizontally), or go up to a client interface of the parent (up again).
- when a $\xrightarrow{Q\_m}$ transition goes horizontally to a sibling sub-component (brother binding), either this one is a primitive, which is sufficient to ensure the lemma, or it is a composite, and there exists necessarily a (delegate) $\xrightarrow{Q\_m}$ reaching down.
- when a $\xrightarrow{Q\_m}$ transition goes down to a sub-component (import binding), similarly, either its a primitive, which is sufficient to ensure the lemma, or it is a composite, and there exists again a (delegate) $\xrightarrow{Q\_m}$ reaching down.

So navigating in the component hierarchy can only go up some steps, then horizontally once to another sibling component, then some steps down in the component hierarchy. The tree forming the component hierarchy being finite, the chain of request transmission is of course finite.

Finally, we can state the following theorem:

Theorem 2 (Correct requests transmission) *In the pNet expressing the semantics of a well-formed and fully connected GCM system, provided the families of future proxies are large enough, every emitted request m is necessarily received by a primitive component.*

This is a direct consequence of Lemma 2. We can prove similar results for the transmission of replies (R_m).

Together with Theorem 1, this shows that deadlocks in a GCM system cannot come from the mechanics of our semantics, but either from a violation of the assumptions of the theorems (well-formedness or full-connection), a problem with the approximation of the model construction (finiteness of queues and proxy families), or a problem in the user application logics, like components not

ready to process a request when needed, typically when several futures would wait for each other.

6.2. Informal comparison with existing semantics

As stated along this paper, we tried to keep the structure of the behavioural semantics as close as possible to the implementation of GCM/ProActive. We also provide a behavioural semantics respecting the operational semantics of ASP [32] and of ASP components [14]. The technicalities and the formal proof of the equivalence between the two semantics, while difficult, could be partially inspired by previous works. In particular [33] proved the equivalence between a higher-level and a lower-level semantics of actors. One strength of this contribution is the compositionality of the actor semantics that could fit closely with our components. However, the differences between the target languages (functional vs. labelled-transitions) make the direct adaptation far from trivial. The semantics of futures and the structural aspects of components should also be adapted to the framework of [33] if we want to reuse it. Also the work of Palmskog et al. [34] could be inspiring to choose how to model a more operational routing of requests in active object languages and prove semantic equivalence in this context (the definition of observational equivalence of [34] could be reused). Overall the previous works dealing with equivalence of actor semantics brought valuable tools and approaches but also showed the technical difficulties that can be raised when formally proving equivalence between two actor semantics.

We provide below a few arguments explaining how the semantics presented here could be related to the one shown in [14] (which is the closest). The structure of the component description is similar to the one presented in this paper and the two notions of well-formed components coincide approximately. Then the semantics is given in terms of transition rules for primitive and composite components, but [14] uses complex structures (maps, stateful objects, functions over structured objects, etc.) for representing component state, requests, and futures instead of labelled transition systems. Proving the equivalence between the semantics will require to exhibit a translation between the two representations of the internal state of the components and proving that each rule of the operational semantics can be simulated by the other semantics, and conversely.

Concerning primitive components there is one rule for sending requests, and one for serving them; there is also one rule for sending and one for receiving request results. Concerning composites, there are three communication rules for each kind of binding (brother, import, export), one rule for a composite emitting a request, one for a composite receiving a result. Concerning request reception, it is handled uniformly by an operator that enqueues a request in the request queue of a component.

Each of the rules is simulated by the coordinated action of several synchronisation vectors and several intermediate pNets. The proof of the lemma above exhibits parts of the arguments that should be used to prove the equivalence, but the exact match between the different rules is way beyond the scope of this paper. It is interesting to notice that among the rules of [14], the ones that correspond to request emission are labelled rules that will be synchronized

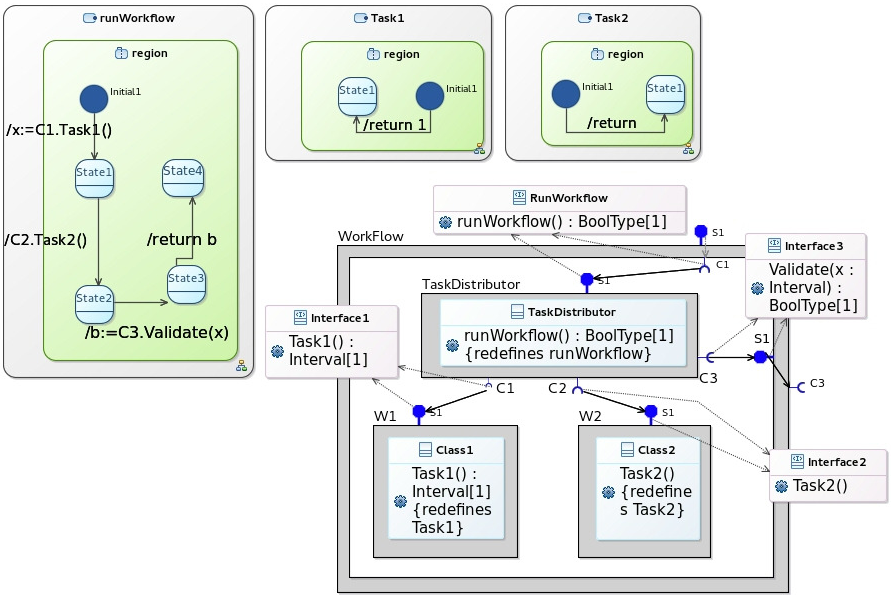

Figure 12: Architecture model for the Workflow application, in VerCors Component Editor

with the arrival of the request in the queue on the other side of the binding in a similar way as what is ensured by the different synchronisation vectors that transmit the requests between pNets. Concerning the flow of futures, it is greatly simplified and abstracted away in [14], but we could refer to [35] for a more operational version of the transmission of the request results, that would fit more closely to the behavioural semantics presented in this article.

6.3. Full example

We will sketch here a small example illustrating the most important of the constructs defined in this article. This example was specified and verified using our VerCors tool-suite; Vercors consists of a graphical editor based on a UML designer, a generator of a behavioural model that implements the semantics specified in this article, and an executable code generator. The generator of the model-checkable behaviour follows the semantics described in the previous section and is done automatically by the Vercors platform [10]. It relies on the CADP toolset to verify correctness properties of the generated behaviour. An overview of the VerCors tool-suite can be found in [31]. Additionally, the graphical editor relies on the static verification of the component architecture [36].

The example is an application called `Workflow`[15], used as a pedagogical

[15] https://github.com/Scale-VerCors/VCEv4/tree/master/Examples/Workflow_01

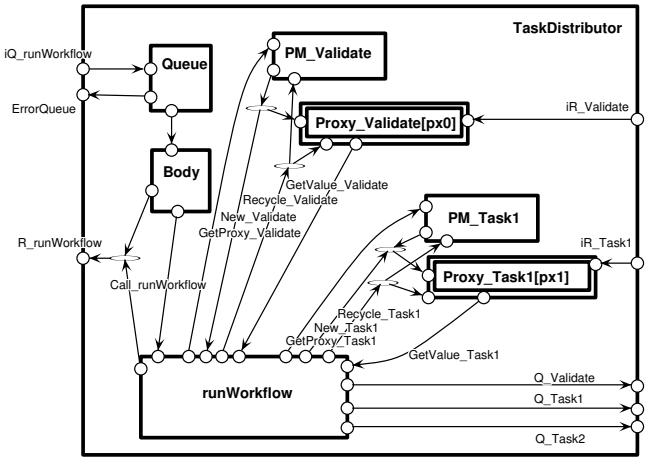

Figure 13: pNet model for the TaskDistributor Primitive component

example to illustrate the notion of futures in GCM. Figure 12 shows the architecture of the application, the signature of interfaces, and the behaviour of its service methods, as shown in our graphical editor. The use-case consists of a workflow execution composite component. This component accepts on its server interface `runWorkflow` requests and uses a `TaskDistributor` primitive component to perform two consecutive tasks handled by two worker components, `W1` and `W2`. Each worker has a single interface receiving tasks to be performed. The `TaskDistributor` is more complex: it receives `runWorkflow` requests on its `S1` interface, and then sequentially calls `Task1` on its client interface `C1` and `Task2` on its interface `C2`. It finally calls a `Validate` request with the result of `Task1` and returns the validation result as a reply to the `runWorkflow` request. The validation step is handled by a component external to the composition.

This example allows us to illustrate the construction of pNets for primitive and composite components, with their queues and bodies, future proxies, service and client delegation processes.

Additionally, this gives us the opportunity to discuss a number of implementation issues, and in particular of simplification of the generated pNet structure depending on the configuration of the GCM components considered. A brute application of the semantic rules from this article would produce an unnecessary number of management pNets, and we give an example of optimisations that are applicable in a significant number of situations. A full description of the implementation of the pNets construction, and of the optimisations, is out of the scope of this article.

6.3.1. Structure of the pNets model

We illustrate here, in an informal manner, the construction of the pNets semantics of the Workflow example. We focus on the pNets hierarchy and their synchronisation vectors, defined graphically. We evaluate the complexity of this

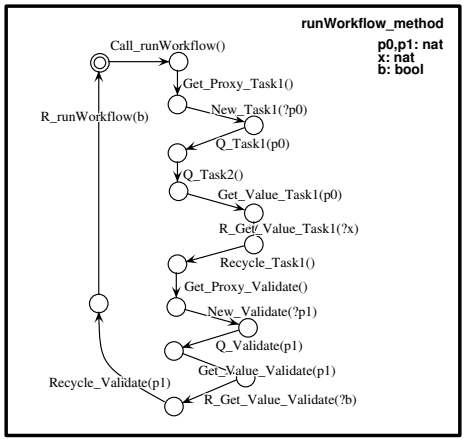

Figure 14: pLTS model for the runWorkflow method

construction.

We start, in Figure 13, with the pNet structure representing the semantics of the `TaskDistributor` primitive component. The pLTSs involved in the composition are not shown here, we only show the pNet structure. This pNet contains:

- The standard `Queue` and `Body` pNets which process the `runWorkflow` request before it is forwarded to the method pLTS.
- The proxy manager and proxy pLTSs for the methods of `C1` and `C3` interfaces that have a return value. The pLTSs treat the requests from `runWorkflow` and forward them to outside of `TaskDistributor`. The only method of `C2` does not return any value, hence, the corresponding proxy and proxy manager are not generated. The requests to *Task2* are simply forwarded to the corresponding worker.
- A pLTS for the `runWorkflow` method which is shown in Figure 14. This is a simple case as the original state machine (in Figure 12) is linear. When comparing the pLTS with the original state machine, notice how the *Get_Proxy**, *New**, *Get_Value**, ..., *Recycle** events have been inserted in the flow.

Each worker is a primitive containing a business process represented here by a `Task` pLTS generated form the corresponding UML state machine.

Figure 15 illustrates the structure of the `Workflow` composite. This part features a complete pNet infrastructure in the composite membrane.

The pNet of the `Workflow` component has three sub-pNets for its internal sub-components, and a full set of pNets modelling its membrane, namely: a `Queue` receiving requests both from its external methods (*iQ_runWorkflow*) and from the `TaskDistributor` sub-pNet (*Q_Validate*). A `Body` dispatches these

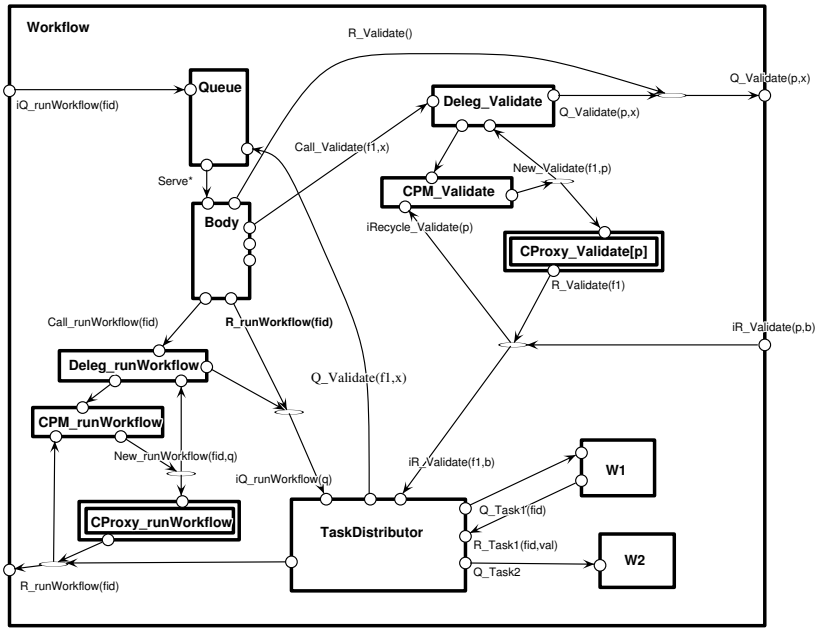

Figure 15: pNet model for the Workflow composite component

requests to the `TaskDistributor` pNet or exports them; in both cases the communication passes through (`Deleg_*`) pLTSs. Because the *Validate* method requires an answer, the pNet has an indexed family of future proxies, managing the *R_Validate* return messages. Similarly, a proxy manager and a proxy family deal with the return of the `runWorkflow` result to the outside.

6.3.2. Properties

From a (finite) pNet model, we have computed the state-space using the CADP verification toolset [37]. For the Workflow use-case, we proved two correctness properties (defined in the Model-Checking Language MCL; we refer to [38] for a description of the languages). MCL formulas are logical formulas using regular expressions, boolean operators, branching modality operators (necessity operator denoted by [ ] and possibility operator denoted by $\langle\ \rangle$), and maximal fixed point operator (denoted by μ).

First, we check that after a call to the `runWorkflow` method it is inevitable that either the result is returned, or the queue of the composite is saturated. We use an inevitability operator in the formula below in order to check that any path in the behaviour graph that starts by *iQ_runWorkflow* will inevitably lead either to *Workflow_ErrorQueue* or to *R_runWorkflow*.

```
['iQ_runWorkflow.*'] inev('Workflow_Errorqueue.*' or 'R_runWorkflow.*')
```

The formula is evaluated to TRUE if the size of the `TaskDistributor` queue is greater or equal to the size of the `Workflow` queue

Second, we show that the distributor can send a `Task2` request before receiving the answer from the `Task1` request. This illustrates that the future mechanism works and provides automatic parallelisation of request treatment. The formula below states that there exists a path in the behavior graph where `Task2` is requested before the result of `Task1` is returned. The model-checker answers TRUE.

```
<'iQ_runWorkflow' . (not 'R_Task1.*')* . 'Q_Task2.*'> true
```

6.3.3. Experimental sizes

This example is very small, compared to other case-studies that we published before [9, 11]. The novelty here is that the input files for the model-checking platform were 100% automatically generated from the mode editors of VerCors.

We give here some figures indicating the size of the model, and the speed of the verification steps. The `W1` and `W2` components pNets have each a single method, a queue and a body. The `TaskDistributor` additionally has 2 proxy managers and $2 * 2$ proxy families. The `Workflow` composite also has a queue and a body, 2 delegate processes, 2 proxy managers, and 2 proxy families. Let p be the number of proxies per proxy family, then we have $(17 + 4 * p)$ basic pNets generated, and also $25 + 2 * p$ (parameterised) synchronisation vectors. For a minimal instantiation of $p = 2$, we obtain 25 basic pNets and 29 parameterised vectors, which is a reasonably small structure. With this proxy size, we also prove that the "Error(NoMoreProxy)" action is not reachable, that proves that this bound is sufficient for all our analyses (see section 4.1.5).

Choosing again minimum values for the domain of parameters to prove our properties, i.e. [0..1] intervals for all data, and for future identifiers, and 4 for the length of request queues, we obtain an LTS with 4 million states and 15 million transitions (noted $(4M/15M)$) for the queue of the composite, reduced to $(9K/65K)$ by branching bisimulation minimisation. The strategy for hiding and reducing intermediate systems is to consider as unobservable all internal communications (marked as "synchronised" in the result of a synchronisation vector) that do not occur in some of the formulas we want to prove. This is the biggest intermediate LTS generated, while most LTSs encoding state machine or controllers have a very small size, with typically less than 30 states. The product LTS of the `TaskDistributor` pNet has a size of (2145/7633), after hiding and minimisation. And the global system product has a size of $(746K/3M)$ before minimisation, that reduces to $(35K/156K)$.

The full verification workflow takes approximatively 6 minutes of processing, when performed sequentially on a Intel Xeon E5-2630 @ 2,6Ghz, running under Fedora 21 (x86-64). The use-case presented here is quite small, however we demonstrated in previous works that the process can scale and be run in a distributed manner. Indeed, many tasks in the hierarchical model generation are independent, and can be run in parallel, with the product and minimisation steps depending on previous tasks, as we demonstrated in [9].

7. Extensions towards Full GCM

Here, we briefly discuss two features from Full GCM, namely the management of component attributes, which allow the implementation of stateful components; and reconfiguration capacities.

Dealing with stateful components

The semantics we presented here does not allow components to have an internal state, however it is easy to add variables to primitive components. Indeed it is sufficient to add an additional sub-pNet to the pNet of the component that stores the value of the variables and accepts *set* and *get* actions. Those actions will be triggered by the service methods (synchronisation vectors for those setters and getters are also necessary).

Such a state management pNet can also be used to express the behaviour of an *attribute controller*. In Fractal, an attribute is a configurable property of a component, it can be accessed and modified by setter and getter functions, exposed outside the component as a non-functional interface. The attribute controller interface is a non-functional interface exposed by the component. Attributes that can be stored and modified can be expressed in pNets by adding the getter and setter functions to the requests that can be enqueued, and by adding synchronisation vectors from the body to the sub-pNet dedicated to state management for getter and setter actions.

About reconfiguration

While we do not show reconfiguration scenarios in this article, we enforce a structure of the models that allows the encoding of reconfiguration primitives. Defining statically the complete structure of the application is often too restrictive. Indeed, especially in a distributed setting, applications must evolve at runtime in order to adapt to changes in the execution environment or to provide improved functionalities. Some component models keep a trace at runtime of the component structure and their dependencies. Knowing how components are composed and being able to modify this composition at runtime provides great adaptation capabilities: the application can be adapted by changing some of the components taking part in the composition or changing the dependencies between the involved components. Reconfigurations consist in changing at runtime the component structure, by adding or removing components in the system, or by changing the way components are bound together. In practice, many component models allow the modification of the architecture at runtime, through a specific set of APIs, like for example Acme [39], SOFA 2.0 [40], Fractal, and GCM. Usually, the set of allowed modifications is constrained by the types, and the architectural style of the involved components.

Verifying reconfigurable component applications is still easier than verifying full fledge object-oriented applications because possible reconfigurations are generally well-identified, and the possible changes can be statically approximated. In the model presented in this article, we keep trace of the component structure when building the model of the application; this not only allows us to build the

model in a compositional manner, but also to encode reconfiguration procedures and to verify the properties of the system when some reconfigurations occur.

Our research report [41] explains how to reason on reconfigurations based on the model presented in this article. The PhD dissertation [10] describes in details the formalisation of non-functional aspects, in particular attribute controllers and interceptors, and collective communications and their reconfiguration.

8. Related Work

Component-based development has in recent years become an established approach. It has shown successes in many application domains such as in distributed and embedded systems. There are several component models that are supplied for building complex systems: Fractal [21], Ptolemy [42], CCM [1, 2, 3], AADL, and GCM [4, 43]. But there are only a few that have a theoretical framework that allows reasoning about modelled systems and verification of their behavioural properties.

Some of the formal developments around components are done with objectives very different from ours. For example, the formalisation of the Fractal model in Alloy [44] brings several interesting properties, but they are mainly related to the model itself more than to the component applications. Similarly, a formal model of GCM has been specified in Isabelle/HOL [14], while it gives an interesting framework to reason on the component model and prove some of its properties, it is not adapted to prove the correct behaviour of applications. A formal framework for reasoning on futures has been defined in [45], but the authors did not provide, to our knowledge, the tools to use their equations in order to automatically or semi-automatically prove properties on programs. Behavioural specification, on the contrary, is better adapted to the correctness proofs for given applications. Among the researches dedicated to the component-oriented behavioural verification that we are aware of, the closest are SOFA, Kmelia, STSLib, and BIP.

The SOFA system [40] is a development and verification framework for large-scale distributed software systems based on hierarchical components. It uses *behaviour protocols* [46, 47] to specify interactions between components in terms of ordering of method invocation events. Behaviour protocols are also used, at each level of the component hierarchy, to define a black box specification of the subsystem. The behaviour compliance and consent relations are defined on behaviour protocols based on their trace semantics, allowing one to prove separately, at each level of the hierarchy, the compliance of an implementation (called architecture) with its specification (protocol). Behaviour protocols can also be encoded e.g. in Promela, that allows for classical LTL model-checking [48].

Kmelia [49, 50, 51] is a component specification model based on the description of complex services. Kmelia and its toolbox COSTO can be used to model software architectures and their properties, these models being later refined to

execution platforms. It can also be used as a common model for studying component or service model properties (abstraction, interoperability, composability), using various verification toolsets, including CADP, MEC5, and Atelier-B. To our knowledge, though, there is no explicit behavioural semantics defined for Kmelia applications.

The STSLib library [52] provides a formal component framework that synthesises components from symbolic protocols in terms of Symbolic Transition Systems (STS). Just as pNets, STS concisely represent infinite systems, however STS rely on Abstract Data Types (ADT) which are more expressive than the Simple Types used in pNets but less intuitive for software engineers. Both formalisms rely on (N-ary) synchronisation vectors, but in STS they are static whereas in pNets they are dynamic. STSLib synthesises components based on their STS protocols; a controller interprets the STS protocol and data from which the ADT is implemented (and generated) in Java. The communication in STS components is rather low level ; both emitter and receiver must agree to exchange a message, and there is no explicit notion of required nor provided services.

On the implementation side, the two approaches are quite different: the implementation of STS simulates the synchronisation vectors that can be expressed in the specification, whereas in our approach, we write only the synchronisation vectors corresponding to the possible communications between components. Our specification language is more independent from the middleware, and it allows us to express complex synchronisations. This allows us to reason on efficient, expressive, and proved communication mechanisms. Overall, even if the pNet formalism is approximately at the same level of abstraction as STS, in our approach, the programmer is rather exposed to a higher-level composition framework, closer to his usual programming and composition concerns.

BIP [53, 54] is a formal framework that allows building and analysing complex component-based systems, both synchronous (reactive) or asynchronous (distributed) by coordinating the behaviour of a set of primitive and heterogeneous components. A component's behaviour is described as a Petri net extended with data and functions, whereas coordination is described as interactions between components and scheduling policies between interactions. Even if the BIP framework allows powerful compositional reasoning on the system, it does not support the definition of hierarchical architectures composing data-sensitive models.

BIP is supported by a toolset including translators from various programming languages as Lustre and C into BIP, a compiler for generating code executable by a dedicated engine, and the verification tool D-Finder. This last tool [55] is not a generic model-checker, but a specific tool for deadlock detection and diagnosis, able to address systems of large size, as shown e.g. in [56, 53].

In the context of the ABS [19] specification language, several verification frameworks exist. ABS is a behavioural specification language that takes the form of a simple object oriented language featuring active objects and futures. Several tools have been developed for the language but three major tools target

program verification.

A precise deadlock analysis [57] has been designed based on behavioural types and resource; and a cost analysis [58] is provided, it is based on static analysis and cost evaluation techniques. Those two works provide verification engines for programs of any size, their limitation reside in the abstraction that is made from the code, which is often precise enough to reason on a wide set of interesting program. Those works are fully automatic but they only reason on one specific property and a new analysis has to be designed if a new property of the program is to be investigated. The advantage of our approach is that it is better adapted to the verification of temporal properties of program where the user specifies the behavioural property of the application that interests her/him.

Probably closer to this article is the Key-ABS framework; Key-ABS [59]. Key-ABS is a framework for reasoning on ABS programs, mostly based on invariant definition and deductive verification techniques. Key-ABS can deal with any kind of functional properties and allows the definition of precise invariant, allowing the user to prove properties on systems of infinite size with a powerful reasoning engine. The main difference is that in Key-ABS the user must have a strong expertise in the programming language and in the definition on invariant annotations which involve the status of a distributed application. The pNets approach allows the automatic generation of a behavioural model and the only expertise from the programmer that is required is to write adequately the properties of his/her application.

Another approach that is very close to the one presented in this paper is the work on the Rebeca system [60, 61]. Rebeca is a Java-like actor programming language which shares a lot of similarities with our language. The Rebeca model has been designed to be at the same time easy to program and easy to verify, particularly using model-checking techniques. While the communication system ensures FIFO communication similarly to GCM/ProActive, the absence of futures and of synchronisation points makes Rebeca easier to analyse and allow the Rebeca verifier to have very good results. Our programming model allows the programmer to write applications with futures ensuring data-driven synchronisation that make programming more natural but also makes building the model and optimising its verification more challenging. Compared to Rebeca models, this article showed how to model futures, transmission of future references, and hierarchical composition of components which is more difficult to model and verify but more compositional and easier to program. Our previous case-studies (see Section 1) somehow show that the verification of these richer systems is still possible.

Our current verification techniques rely on model-checking which limits us concerning the size of the verifiable system, but the parameterised nature of pNets is particularly adapted to the use of SMT solvers, which would allow us to reason on systems of infinite size. The use of SMT solvers on pNets is still a mid-term perspective, but it would make our results even easier to compare with the ABS tools.

9. Conclusion

This article provides a formal framework for the generation of behavioural semantics of asynchronous distributed software components. Asynchronous software components provide a convenient programming abstraction for designing large-scale distributed systems, where each component acts as an autonomous entity, only communicating with the others using requests and replies. This article describes the behavioural semantics of such components. Behavioural semantics enable the generation of a model of the program behaviour; then the correctness of the application can be verified using dedicated platforms, for example based on model-checking techniques. The main contributions of this article are:

- a minimal formal definition of pNets. This definition does not cover all aspects previously published, but constitutes a simple core definition, self-contained and sufficient for this article,
- a formal definition of GCM components, their abstract syntax, and an informal semantics,
- a precise formal definition of the behavioural semantics of GCM, in the form of structural rules constructing pNets, these rules encode hierarchical component composition, asynchronous requests, and futures into pNets,
- arguments validating our approach by showing that each action emitted by a component is synchronised at the upper level,
- an example illustrating the approach and showing the practical use of the behavioural semantic rules.

Among these contributions, the main part concerns the behavioural specification of the most important Fractal/GCM features. For some of these, we have already described a behavioural semantics in previous conference papers, but only in an informal way. Here we have given a full definition, and a procedure for building their pNet models. This includes the basic structure for GCM components, request queues, body expressing the service policy, future proxies and proxy managers; it also defines the semantic artefacts needed for first class futures.

The example described in the last section features all the aspects listed above, except for first class futures. We also use this example to comment on the complexity of our semantic model generation.

A first version of the VerCors specification and verification platform is now available [31]; it implements the semantics provided in this article. The example of Section 6.3 has been entirely generated by the platform. We also plan to extend this work in several directions. First there are still some GCM features that are not included in the current core semantics, and that are important in practical applications. This is the case of collective interfaces (see [4]), and reconfiguration procedures, either considering explicit reconfiguration scripts,

or autonomic components. Some of these features have already been partially investigated in [41].

The approach presented in this article can also be applied to other types of distributed languages or formalisms, in the way we have dealt with Active objects, Fractal components, and GCM. pNets encode in a very flexible and expressive way any kind of process composition, communication, and synchronisation, provided it stays within the family of first-order hierarchical structures. This rules out formalisms such as the π-calculus or the chemical machine, but includes other process calculi *à la* CCS, and most component models (CCM, SCA, Creol, ...) and their implementations. Indeed, the core semantics presented here can be mostly re-used outside the strict scope of ProActive/GCM. First, the notions related to component composition are slightly different from one component model to another but rely on the same bases, adapting the component structure to another component model like SCA or CCM for example would raise no major difficulty. Concerning the execution model, active objects are some form of actors (which is probably the programming model the most adapted to asynchronous distributed components) and adapting our semantics to other actor models (e.g. Erlang, Salsa, Rebeca, ABS, Creol, Akka) would require slight changes in the pNet generation rules to reflect the slightly different operational semantics of those languages. Adapting our model to tightly coupled components communicating more synchronously than the ones presented here (e.g. MPI, BSP) would be more difficult.

In another direction, we plan to relate the behavioural semantics of this article with the more "operational" one defined in [13]. We have used the latter in our research on theorem-prover based formalisations and proofs for GCM, and relating the two semantics would give us powerful tools for dealing with properties of dynamically evolving applications, and for combining model-checking and theorem proving methods to reason about realistic applications.

Acknowledgements. This work was partially funded by the French FSN project OpenCloudware (Fond national pour la Société Numérique).

References

[1] CCA forum, The Common Component Architecture (CCA) Forum home page, http://www.cca-forum.org/ (2005).

[2] Object Management Group, Inc. (OMG), CORBA Component Model Specification, OMG Headquarters Edition, `http://www.omg.org/cgi-bin/apps/doc?formal/06-04-01.pdf` (April 2006).

[3] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, SCA Service Component Architecture, Assembly Model Specification, Tech. rep., OSOA (March 2007).

[4] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, C. Pérez, GCM: A Grid Extension to Fractal for Autonomous Distributed Components, Annals of Télécommunications 64 (1-2) (2009) 5–24.

[5] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, E. Madelaine, Behavioural Models for Distributed Fractal Components, Annals of Télécommunications 64 (1-2) (2009) 25–43.

[6] L. Henrio, E. Madelaine, M. Zhang, A Theory for the Composition of Concurrent Processes, in: E. Albert, I. Lanese (Eds.), 36th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Vol. LNCS-9688 of Formal Techniques for Distributed Objects, Components, and Systems, Heraklion, Greece, 2016, pp. 175–194. doi:10.1007/978-3-319-39570-8_12.
URL `https://hal.inria.fr/hal-01432917`

[7] T. Barros, L. Henrio, E. Madelaine, Verification of Distributed Hierarchical Components, in: International Workshop on Formal Aspects of Component Software (FACS'05), Electronic Notes in Theoretical Computer Science (ENTCS), Macao, 2005.

[8] A. Cansado, L. Henrio, E. Madelaine, Transparent First-class Futures and Distributed Component, in: International Workshop on Formal Aspects of Component Software (FACS'08), Malaga, 2008.

[9] R. Ameur-Boulifa, R. Halalai, L. Henrio, E. Madelaine, Verifying Safety of Fault-Tolerant Distributed Components, in: International Workshop on Formal Aspects of Component Software (FACS'11), Oslo, 2011.

[10] O. Kulankhina, A framework for rigorous development of distributed components: formalisation and tools, Ph.D. thesis, Université de Nice Sophia Antipolis (Octobre, 2016).

[11] N. Gaspar, L. Henrio, E. Madelaine, Formally Reasoning on a Reconfigurable Component-Based System - A Case Study for the Industrial World, in: International Symposium on Formal Aspects of Component Software (FACS 2013), Lecture Notes in Computer Science, Springer, Nanchang, China, 2013.

[12] R. A. Boulifa, L. Henrio, E. Madelaine, Behavioural models for group communications, in: WCSI-10: International Workshop on Component and Service Interoperability, Malaga, Spain, 2010.

[13] L. Henrio, F. Kammüller, M. Rivera, An Asynchronous Distributed Component Model and Its Semantics, in: F. de Boer, M. Bonsangue, E. Madelaine (Eds.), FMCO'08, Vol. 5751 of LNCS, Springer, Heidelberg, 2008, pp. 159–179.

[14] L. Henrio, F. Kammüller, M. U. Khan, A Framework for Reasoning on Component Composition, in: FMCO 2009, Lecture Notes in Computer Science, Springer, 2010.

[15] E. Bruneton, T. Coupaye, M. Leclerc, V. Quéma, J.-B. Stefani, The Fractal Component Model and Its Support in Java, Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems 36 (11-12).

[16] E. B. Johnsen, O. Owe, I. C. Yu, Creol: A types-safe object-oriented model for distributed concurrent systems, Journal of Theoretical Computer Science 365 (1–2) (2006) 23–66.

[17] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, W. D. Meuter, Ambient-Oriented Programming in AmbientTalk, in: D. Thomas (Ed.), ECOOP, Vol. 4067 of Lecture Notes in Computer Science, Springer, 2006, pp. 230–254.

[18] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing Active Objects to Concurrent Components, ECOOP 2010 – Object-Oriented Programming (2010) 275–299.

[19] E. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A Core Language for Abstract Behavioral Specification, in: Formal Methods for Components and Objects, Vol. 6957 of LNCS, Springer Berlin Heidelberg, 2012, pp. 142–164. doi:10.1007/978-3-642-25271-6.

[20] R. Ziaei, G. Agha, Synchnet: A petri net based coordination language for distributed objects, in: F. Pfenning, Y. Smaragdakis (Eds.), GPCE, Vol. 2830 of Lecture Notes in Computer Science, Springer, 2003, pp. 324–343.

[21] E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J. B. Stefani, An Open Component Model and Its Support in Java, in: 7th Int. Symp. on Component-Based Software Engineering (CBSE-7), LNCS 3054, 2004.

[22] E. Bruneton, T. Coupaye, J. B. Stefani, The Fractal Component Model, Tech. rep., ObjectWeb Consortium, `http://fractal.objectweb.org/specification/index.html` (February 2004).

[23] G. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1) (1997) 1–72.
URL `citeseer.nj.nec.com/article/agha97foundation.html`

[24] F. Mattern, S. Fünfrocken, A non-blocking lightweight implementation of causal order message delivery, in: K. P. Birman, F. Mattern, A. Schiper (Eds.), Theory and Practice in Distributed Systems: International Workshop Dagstuhl Castle, Germany, September 5–9, 1994 Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 197–213.

[25] B. Charron-Bost, F. Mattern, G. Tel, Synchronous, asynchronous, and causally ordered communications, Distributed Computing 9:173–191.
URL citeseer.ist.psu.edu/charron-bost95synchronous.html

[26] C. A. Varela, Programming Distributed Computing Systems: A Foundational Approach, The MIT Press, 2013.

[27] D. Caromel, L. Henrio, B. Serpette, Asynchronous and deterministic objects, in: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 2004, pp. 123–134, iSBN 1-58113-729-X.

[28] L. Henrio, Formal models for programming and composing correct distributed systems, Ph.D. thesis, Université de Nice Sophia-Antipolis, HDR Thesis (Jul. 2012).

[29] L. Henrio, E. Madelaine, M. Zhang, pNets: an Expressive Model for Parameterised Networks of Processes, in: Formal Approaches to Parallel and Distributed Systems (4PAD)-Special Session of Parallel, Distributed and network-based Processing (PDP), Turku, Finland, 2015, extended version in https://hal.inria.fr/hal-01055091.
URL https://hal.inria.fr/hal-01139432

[30] R. Cleaveland, J. Riely, Testing-Based Abstractions for Value-Passing Systems, in: J. P. B. Jonsson (Ed.), Int. Conf. on Concurrency Theory (CONCUR'94), Vol. 836 of LNCS, Springer, Heidelberg, 1994, pp. 417–432.

[31] L. Henrio, O. Kulankhina, S. Li, E. Madelaine, Integrated Environment for Verifying and Running Distributed Components, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 66–83. doi:10.1007/978-3-662-49665-7_5.

[32] D. Caromel, L. Henrio, A Theory of Distributed Objects, Springer-Verlag New York, Inc., 2005.

[33] I. A. Mason, C. L. Talcott, Actor languages their syntax, semantics, translation, and equivalence, Theoretical Computer Science 220 (2) (1999) 409 – 467. doi:http://dx.doi.org/10.1016/S0304-3975(99)00009-2.

[34] M. Dam, K. Palmskog, Location-independent routing in process network overlays, Serv. Oriented Comput. Appl. 9 (3-4) (2015) 285–309. doi:10.1007/s11761-014-0173-7.
URL http://dx.doi.org/10.1007/s11761-014-0173-7

[35] L. Henrio, M. U. Khan, Asynchronous components with futures: Semantics and proofs in isabelle/hol, in: Proceedings of the Seventh International Workshop, FESCA 2010, ENTCS, 2010.

[36] L. Henrio, O. Kulankhina, D. Liu, E. Madelaine, Verifying the correct composition of distributed components: Formalisation and Tool, in: FOCLASA, Rome, Italy, 2014.
URL https://hal.inria.fr/hal-01055370

[37] H. Garavel, F. Lang, R. Mateescu, An overview of CADP 2001, European Association for Software Science and Technology Newsletter 4 (2002) 13–24.

[38] R. Mateescu, D. Thivolle, A Model Checking Language for Concurrent Value-Passing Systems, in: K. S. J. Cuellar, T. S. E. Maibaum (Ed.), FM'08, Vol. 5014 of LNCS, Springer, Heidelberg, 2008.

[39] D. Garlan, R. T. Monroe, D. Wile, Acme: Architectural description of component-based systems, Foundations of component-based systems 68 (2000) 47–68.

[40] T. Bures, P. Hnetynka, F. Plasil, SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, in: Proceedings of SERA 2006, IEEE CS, 2006, pp. 40–48.

[41] R. Ameur-Boulifa, L. Henrio, E. Madelaine, A. Savu, Behavioural Semantics for Asynchronous Components, Research Report RR-8167, INRIA (Dec. 2012).
URL http://hal.inria.fr/hal-00761073

[42] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, Y. Xiong, Taming Heterogeneity - the Ptolemy Approach, Proceedings of the IEEE 91 (1) (2003) 127–144.

[43] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, C. Zoccolo, Autonomic Grid Components: the GCM Proposal and Self-optimising ASSIST Components, in: Joint Workshop on HPC Grid programming Environments and COmponents and Component and Framework Technology in High-Performance and Scientific Computing at HPDC'15, 2006.

[44] P. Merle, J.-B. Stefani, A formal specification of the Fractal component model in Alloy, Research Report RR-6721, INRIA (2008).
URL http://hal.inria.fr/inria-00338987/en/

[45] F. S. de Boer, D. Clarke, E. B. Johnsen, A Complete Guide to the Future, in: R. De Nicola (Ed.), Programming Languages and Systems, Vol. 4421 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 316–330. doi:10.1007/978-3-540-71316-6.

[46] F. Plasil, S. Visnovsky, Behavior protocols for software components, IEEE Transactions on Software Engineering 28 (11) (2002) 1056–1076. doi:10.1109/TSE.2002.1049404.

[47] T. Poch, O. Sery, F. Plasil, J. Kofron, Threaded behavior protocols, Formal Aspects of Computing 25 (4) (2013) 543–572. doi:10.1007/s00165-011-0194-3.

[48] J. Kofron, Checking Software Component Behavior Using Behavior Protocols and Spin, in: proceedings of Applied Computing 2007, Seoul, Korea, 2007.

[49] P. André, G. Ardourel, C. Attiogbé, Adaptation for hierarchical components and services, Electron. Notes Theor. Comput. Sci. 189 (2007) 5–20. doi:http://dx.doi.org/10.1016/j.entcs.2007.05.045.

[50] C. Attiogbé, P. André, G. Ardourel, Checking Component Composability, in: 5th International Symposium on Software Composition (ETAPS/SC'06), Vol. 4089 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[51] P. André, G. Ardourel, C. Attiogbé, Composing Components with Shared Services in the Kmelia Model, in: 7th International Symposium on Software Composition, SC'08, Vol. 4954 of LNCS, Springer, 2008.

[52] F. Fernandes, J.-C. Royer, The STSLib project: Towards a formal component model based on STS, Electronic Notes in Theoretical Computer Science 215 (2008) 131–149.

[53] S. Bensalem, M. Bozga, T.-H. Nguyen, J. Sifakis, Compositional verification for component-based systems and application, IET Software 4 (3) (2010) 181–193.

[54] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, J. Sifakis, Rigorous Component-Based System Design Using the BIP Framework, IEEE Software 28 (3) (2011) 41–48.

[55] S. Bensalem, M. Bozga, T.-H. Nguyen, J. Sifakis, D-Finder: A Tool for Compositional Deadlock Detection and Verification , in: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, Vol. 5643 of Lecture Notes in Computer Science, Springer, 2009, pp. 614–619.

[56] A. Basu, M. Gallien, C. Lesire, T.-H. Nguyen, S. Bensalem, F. Ingrand, J. Sifakis, Incremental component-based construction and verification of a robotic system, in: ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings, Vol. 178 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2008, pp. 631–635.

[57] E. Giachino, C. Laneve, M. Lienhardt, A framework for deadlock detection in ABS, Journal of Software and Systems Modeling (2015) 1–36.

[58] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, G. Román-Díez, SACO: Static analyzer for concurrent objects, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 8413 of LNCS, Springer Berlin Heidelberg, 2014, pp. 562–567. doi:10.1007/978-3-642-54862-8_46.
URL http://dx.doi.org/10.1007/978-3-642-54862-8_46

[59] C. C. Din, R. Bubel, R. Hähnle, KeY-ABS : A deductive verification tool for the concurrent modelling language ABS, in: Proceedings of the 25th International conference on Automated Deduction (CADE 2015), Springer, 2015, pp. 517–526.

[60] M. Sirjani, F. S. de Boer, A. Movaghar, A. Shali, Extended rebeca: A component-based actor language with synchronous message passing, in: Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France, IEEE Computer Society, 2005, pp. 212–221. doi:10.1109/ACSD.2005.12.
URL http://dx.doi.org/10.1109/ACSD.2005.12

[61] M. Sirjani, A. Movaghar, A. Shali, F. S. de Boer, Model checking, automated abstraction, and compositional verification of rebeca models, J. UCS 11 (6) (2005) 1054–1082. doi:10.3217/jucs-011-06-1054.
URL http://dx.doi.org/10.3217/jucs-011-06-1054

Appendix A. An operational semantics for pNets

This appendix provides an operational semantics for the pNet model; it is based on a valuation domain for the variables of the pNet, that can be finite, infinite, or even contain new variables.

To give a semantics to pNets, we need a unique *valuation domain* $\mathcal{D}$. This domain can possibly be a countable instantiation domain for each variable. To simplify the semantics, we require that it is possible to decide whether a boolean expression in $\mathcal{D}$ is true, and to decide whether two expressions have the same value (e.g. when two action labels are the same). If we choose a finite domain for each variable and if each pLTS has a finite set of states, the semantics of the pNet will be a finite LTS that can be used in a finite-state model-checker.

pNets semantics is expressed as a labelled transition system with one state for each potential state of the pNet composition; each state correspond to the composition of the states of all the pLTSs involved in the composition, i.e. all the leaves of the composition tree. Such states are denoted $\triangleleft s_i^{i\in I} \triangleright$ where I corresponds to the indexes of the leaves. Transition between these nodes are constructed by composing the synchronisation vectors that allow two pLTSs to communicate.

We let $\phi = \{x_j \to V_j | j \in J\}$ be a valuation function where x_j range over variables of the considered pNet (each variable must be given a value), and $V_j \in \mathcal{D}$. Such a valuation maintains a mapping from variables to values. For a term $t \in \mathcal{T}$, $t\phi \in \mathcal{D}$ is the value of the term obtained by replacing each variable by their values given by ϕ. A valuation can be applied to expressions, actions, or even indexed sets. In all cases, the variables are replaced by their value and the new expressions are evaluated. The set of valuation functions, Φ, allows the precise definition of the state-space to be considered: only valuation functions such that $\phi \in \Phi$ are considered. We define an update operator $+$ on valuations, where $\phi_1 + \phi_2$ replaces some of the values defined in ϕ_1 by the ones in ϕ_2; also ϕ_2 might define new entries, formally:

$$\{x_j \to V_j | j \in J\} + \{x'_j \to V'_j | j \in J'\} = \{x'_j \to V'_j | j \in J'\} \cup \{x_j \to V_j | j \in J \setminus J'\}$$

Note that variables could be used locally to each pNet/pLTS, it is thus possible to use qualified names to avoid collision of variable names in the valuation. To simplify notations, in the semantics we suppose that variable names are unique.

Definition 5 (Operational semantics of closed pNets). Let $\phi_0 \in \Phi$ be an initial valuation associating a value to each variable of P. The semantics of a pNet *pNet* is an LTS $\|pNet\|_\Phi = \langle\!\langle S_\Phi(pNet), S_0(pNet), \to \rangle\!\rangle$ where:

- states are (fully valuated) hierarchical composition of product states of the sub-pNets, more precisely states are $S_\Phi(pNet)$, constructed as:

$$S_\Phi(\langle\!\langle S, s_0, L, \to \rangle\!\rangle) = \{(s, \phi) | s \in S \ \wedge \ \ \phi \in \Phi\}$$
$$S_\Phi(\langle\!\langle pNet_i^{i\in I}, SV_k^{k\in K} \rangle\!\rangle) = \{\triangleleft s_i^{i\in I\phi} \triangleright \mid \phi \in \Phi \wedge \forall i \in I\phi.\, s_i \in S_\Phi(pNet_i)\}$$
$$S_\Phi(Queue(M)) = \{(M_j\phi_j)^{j\in[1..n]} | n \in \mathbb{N} \ \wedge \ \forall j.\, (M_j \in M \wedge \phi_j \in \Phi)\}$$

- The initial state $S_0(P)$ of P is the composition of initial states:

$$S_0(\langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle) = (s_0, \phi_0)$$
$$S_0(\langle\!\langle pNet_i^{i\in I}, SV_k^{k\in K} \rangle\!\rangle) = \lhd S_0(pNet_i) \rhd^{i \in I\phi_0}$$
$$S_0(Queue(M)) = [\,]$$

- labels are $\{\alpha\phi | \alpha \in \mathrm{Sort}(pNet) \wedge \phi \in \Phi\}$;

- and the set of transition is denoted $[\![pNet]\!]_\Phi$, it is defined as the smallest set verifying the rules below.

$$\frac{\begin{array}{c}\phi, \phi', \phi'' \in \Phi \\ s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J} \rangle} s' \in \rightarrow \qquad iv(\alpha) = x_i^{i\in I} \qquad \forall i \in I.\, V_i \in \mathcal{D} \\ e_b\phi' = True \qquad \phi' = \phi + \{x_i \rightarrow V_i\} \qquad \phi'' = \phi' + \{x_j \rightarrow e_j\phi' | j \in J\}\end{array}}{(s,\phi) \xrightarrow{\alpha\phi\{(?x_i \leftarrow Vi)^{i\in I}\}} (s', \phi'') \in [\![\langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle]\!]_\Phi} \textbf{Tr1}$$

$$\frac{\begin{array}{c}\phi, \phi' \in \Phi \qquad \alpha_j^{j\in J} \rightarrow \alpha \in SV_k^{k\in K} \qquad \forall i \in I\phi \setminus J\phi.\, s_i' = s_i \\ \forall j \in J\phi.\, \phi_j \in \Phi \wedge s_j \xrightarrow{\alpha_j\phi_j} s_j' \in [\![pNet_j]\!]_\Phi \qquad \phi' = \phi + \uplus\{\phi_j\}^{j\in J\phi}\end{array}}{\lhd s_i^{i\in I\phi} \rhd \xrightarrow{\alpha\phi'} \lhd s_i'^{i\in I\phi'} \rhd \in [\![\langle\!\langle pNet_i^{i\in I}, SV_k^{k\in K} \rangle\!\rangle]\!]_\Phi} \textbf{Tr2}$$

$$\frac{n \in \mathbb{N} \qquad \forall j \in [1..n+1].\, M_j \in M \wedge \phi_j \in \Phi}{(M_j\phi_j)^{j\in[1..n]} \xrightarrow{Q\_M_{n+1}\phi_{n+1}} (M_j\phi_j)^{j\in[1..n+1]} \in [\![Queue(M)]\!]_\Phi} \textbf{Tr3}$$

$$\frac{n \in \mathbb{N} \qquad \forall j \in [1..n].\, M_j \in M \wedge \phi_j \in \Phi}{(M_j\phi_j)^{j\in[1..n]} \xrightarrow{Serve\_M_1\phi_1} (M_{j+1}\phi_{j+1})^{j\in[1..n-1]} \in [\![Queue(M)]\!]_\Phi} \textbf{Tr4}$$

Note that states of pLTSs and Queues are associated with a valuation, but states of the pNets also use the valuation to decide (expand) the set of sub-pNets embedded in the pNet.

Rule **Tr1** deals with transitions between states of the pLTS. The resulting action is obtained by replacing input variables by values that can be received (in $\mathcal{D}$); ϕ is also applied to evaluate other parameters, i.e. output expressions. Only input variables and assigned variables change value in the valuation; the resulting valuation is obtained by the successive updates from the input variable assignments, and the explicit assignments from the transition. Remark that the predicate in a transition is evaluated in a valuation that includes the values carried by the communication action, allowing for expressing non-local decisions as in Lotos *gate negotiation*.

Rule **Tr2** deals with transitions between states of the pNet; the resulting valuation is obtained by the combination of all updates happening in the sub-pNets involved in the synchronisation ($\phi \uplus \{\phi_j\}^{j\in J\phi}$ in the rule); sub-pNets not involved in the transition keep the same state. Note that, as pLTSs have disjoint sets of variables, the domain of each valuation ϕ_j should be distinct.

Rule **Tr3** and **Tr4** defines a simple FIFO behaviour of the Queues.

Appendix B. Signature of Behavioural Semantics

The table below shows the signatures of the functions computing the behavioural semantics presented here:

Function	Signature	Description	page
$[\![\ ]\!]$	$Component \to pNet$	basic behavioural semantics	23, 31
$[\![\ ]\!]^{F1}$	$Component \to pNet$	behavioural semantics for first-class futures	40
$[\![\ ]\!]_{service}$	$MethodLabels \times \mathcal{P}(MSignature \times Desc) \to pNet$	Service methods(not specified here)	
$[\![\ ]\!]_{body}$	$\mathcal{P}(MethodLabels) \times \mathcal{P}(MethodLabels) \to pNet$	The body: serves requests in a FIFO order	24
$[\![\ ]\!]_{proxyManager}$	$MethodLabels \to pNet$	Manages future proxies	26
$[\![\ ]\!]_{proxy}$	$MethodLabels \to pNet$	future proxy	26
		overloaded for composite components	32
	$MethodLabels \times GRef \to pNet$	overloaded for first-class futures	40
$[\![\ ]\!]_{FutDetect}$	$MethodLabels \to pNet$	pLTS detecting a future received as request parameter	40
$[\![\ ]\!]_{delegate}$	$MethodLabels \to pNet$	Delegation method (in composite components)	32

Appendix C. Proof of the Complete Synchronisation Theorem

Let

$$W \ = \ \{Q\_m_i(arg) \,|\, m_i \text{ is never invoked}\}$$

In order to prove the theorem, one must consider an induction on the structure of the pNet $[\![C]\!]$. And, at each structural level, i.e. in each pNet *pNet* included in $[\![C]\!]$, we prove that:

- The set of actions emitted by sub-pNets of *pNet* that is not synchronised by the synchronisation vectors of *pNet* is included in W (first line of the Definition 4, called **case 1** in the following).
- Considering the set of actions in the sort of *pNet* that is not synchronised because the synchronisation vector is never triggered; this set is included in W (second line of the Definition 4, called **case 2** in the following).

The inclusion of the last line of Definition 4 in W is guaranteed by the fact that we consider each *pNet* included in $[\![C]\!]$ recursively (including the top level). It is sufficient to prove that, at each level, the unsynchronised actions are included in W to ensure that the recursive union of those unsynchronised actions is included in W.

Overall, the proof consists in considering all labels of all sorts of all pNets *pNet* in $[\![C]\!]$, and showing that each such action is not in $Unsynchronised([\![C]\!])$

or is in W. Also for pLTS, it must be checked that each action appears on at least one label.

The proof can be organised as a double case analysis, first enumerating existing labels, then checking that each such label is synchronised by a synchronisation vector, every time it can be emitted. Note that TauMonitor synchronises implicitly all actions that are of the form $\underline{a}$ (section 3.3). Thus we only have to consider, in the synchronisation vectors, actions that are not synchronised actions. Those actions are emission and reception actions of the following labels for the base semantics: Q_m_i, R_m_i, $Serve\_m_i$, $Call\_m_i$, New_m_i, $GetValue\_m_i$, $Recycle\_m_i$, $GetProxy\_m_i$. Then one should also consider *Forward* actions for first class futures.

Note that in general, concerning arguments, it is easy to check that for each variable present in the synchronisation vectors one single action defines the value of this variable and the other instances of the variable are (directly or indirectly) matched with input variables of pLTSs. However the case is a little more complicated concerning future indexes: a future proxy with the right index should exist to allow the synchronisation vector to be triggered.

In this proof sketch, we will focus on only some of the labels: we detail the cases for Q_m_i (in any rule of the base semantics), and sketch the proof for iQ_m_i (other cases are similar or simpler).

- $Q\_m_i(p, args)$ **in the semantics of a primitive component**

$C = CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >$. Only service methods can emit $Q\_m_i(p, args)$, thus only the service methods must be considered for the **case 1**. For **case 2**, we only have to check that $Q\_m_i(p, args)$ is indeed emitted by $[\![C]\!]$ as a result of an emission by a service method.

  - Q_m_i is emitted by a service method $\mathcal{SM}_l = [\![m_l, M_k^{k\in K}]\!]_{service}$ of a server interface $SItf_n$ then there must be a client interface $CItf_j \in CItf_j^{j\in J}$ such that $m_i \in \text{MethLabel}(CItf_j)$. Thus, Rule [P2.3] defines the following synchronisation vector, ensuring the synchronisation of the action Q_m_i emitted by the service method:

$$\langle -, -, l{\mapsto}Q\_m_i(p, arg), -, - \rangle \rightarrow Q\_m_i(p, arg)$$

  - $Q\_m_i \in Sort([\![C]\!])$, by definition of $[\![C]\!]$ the only synchronisation triggering Q_m_i is the one above, if the method is invoked by at least one service method then the existence of the preceding synchronisation vector is sufficient to conclude; else $Q\_m_i \in W$.

- $Q\_m_i(p, args)$ **in the semantics of a composite**

$C = CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, \overline{Binding} >$. For **case 1**, Q_m_i can be sent either by the service methods, i.e. the predefined $Deleg\_m_i$ pLTSs, or by sub-components.

  - Concerning service methods, two cases are possible depending whether it is a method of a client interface or of a server interface (note that the name of the service method emitting Q_m_i is m_i).

For client interfaces, Rule [C3] specifies the following synchronisation vector for requests emitted by the component:

$$\langle -, R\_m_i(f), i{\mapsto}Q\_m_i(p, arg), -, -, - \rangle \rightarrow Q\_m_i(p, arg)$$

It is necessary to check that the body sup-pNet emits the action $R\_m_i(f)$, i.e. that $R\_m_i(f) \in Sort([\![m_l^{l\in L}]\!]_{body})$, which is true because $m_i \in m_l^{l\in L}$.

The case above deals with the only synchronisation vector emitting requests, i.e. the only case where **case 2** has to be considered. The argument above ensures that each of the synchronisation vector of $[\![C]\!]$ emitting Q_m_i can be triggered and thus each Q_m_i in $Sort([\![C]\!])$ is synchronised.

Concerning methods of the server interfaces; consider a server interface $SItf_j$ and a method $m_i \in \text{MethLabel}(SItf_j)$. Rule [C5.1] defines the following synchronisation vector that deals with Q_m_i :

$$\langle -, R\_m_i(f), i{\mapsto}Q\_m_i(q, arg), -, -, k{\mapsto}iQ\_m'_i(q, arg) \rangle \rightarrow \underline{Q\_m_i}(q, arg)$$

Indeed, as the component is complete, there must be a binding between the internal interface $SItf_j$ of the composite and a server interface of a sub-component, i.e. $(CName.\text{Name}(SItf_j), Comp_k.SI_2) \in \overline{Binding}$. Additionally, by definition of well-formed components, the type SI_2 must be a subtype of the type of $SItf_j$; consequently $m'_i \in \text{MethLabel}(SItf_2)$ (where $SItf_2$ is the interface named SI_2 and $m'_i = m_i\{\!\{\text{Name}(SItf_j) \leftarrow SI_2\}\!\}$) and $iQ\_m'_i(q, arg) \in Sort([\![Comp_k]\!])$. Finally, as $m_i \in m_l^{l\in L}$, we have $R\_m_i(f) \in Sort([\![m_l^{l\in L}]\!]_{body})$. This shows that the synchronisation vector can be triggered.

- Concerning request emission by sub-components, there is a client interface $CItf$ of $Comp_k$, such that $m_i \in \text{MethLabel}(CItf)$. As the component is fully connected, the client interface must be connected, two cases are possible:
  - Either there is a server interface $SItf$ of another sub-component $Comp_{k'}$ such that $CItf$ is bound to $SItf$ in bindings: $(Comp_k.\text{Name}(CItf), Comp_{k'}.\text{Name}(SItf)) \in \overline{Binding}$. In that case, Rule [C7.1] defines the following synchronisation vector that synchronises Q_m_i:

    $$\langle -, -, -, -, -, (k{\mapsto}Q\_m_i(f, arg), k'{\mapsto}iQ\_m'_i(f, arg)) \rangle \rightarrow \underline{Q\_m_i}(f, arg)$$

    Additionally, because of subtyping entailed by bindings we have $m'_i \in \text{MethLabel}(SItf_2)$ (where $m'_i = m_i\{\!\{\text{Name}(CItf) \leftarrow \text{Name}(SItf)\}\!\}$) and $iQ\_m'_i(f, arg)) \in \text{Sort}([\![Comp_{k'}]\!])$. The synchronisation vector can thus be triggered.
  - Else, the client interface must be connected to an interface $SItf$ of the component C: $(Comp_k.\text{Name}(CItf), \text{Name}(C).\text{Name}(SItf)) \in \overline{Binding}$. In that case, Rule [C6.1] synchronises the request emission

with the acceptance of the request by the queue of C, according to the following synchronisation vector:

$$\langle iQ\_m_i''(f, arg), -, -, -, -, k{\mapsto}Q\_m_i(f, arg)\rangle \rightarrow \underline{Q\_m_i''}(f, arg)$$

where $m_i'' = m_i\{\!\{\text{Name}(CItf) \leftarrow \text{Name}(SItf)\}\!\}$. Because of sub-typing, the queue must accept the incoming request.

- **The case of** iQ_m_i

Showing that iQ_m_i actions are synchronised is mostly similar to the cases above. iQ_m_i actions received by the queue are always synchronised. We use here the fact that we forbid sub-typing Indeed, in a composite if iQ_m_i can be received by a sub-pNet corresponding to a sub-component, then this request invocation may not be sent by the sub-component connected to it for sub-typing reasons. In the conditions stated in the theorem however, this case raises no particular difficulty.